

It's not every day that three world technology leaders join together to define a new standard in computing architecture. But that's exactly what Apple, IBM, and Motorola have done since combining forces in the PowerPC alliance. At the Somerset design facility in Austin, Texas, engineers from each of the three companies design PowerPC microprocessors, employing a formal VLSI design methodology drawn from the best of IBM's and Motorola's development environments. With designs completed on schedule for the first two PowerPC family members—the PowerPC 601 and PowerPC 603 microprocessors—the computing industry can expect a steady stream of PowerPC microprocessors satisfying a broad range of market requirements to be introduced in the years to come.

This month's issue of *Communications* covers a wide range of topics surrounding the PowerPC technology, beginning with a description by Moore and Stanphill of the origins of the Apple-IBM-Motorola alliance itself and an overview of the first generation of four distinct microprocessors designed at Somerset.

The PowerPC architects paid particular attention to defining PowerPC Architecture details, making it flexible and scalable, thus providing plenty of room for growth across a varied family of processors suitable for a wide range of designs—from high-performance, multiprocessing systems to handheld computing devices. In addition, the definition of the architecture ensures a guaranteed common software environment for operating system and applications software developers. Diefendorff describes the history of PowerPC Architecture evolution.

Design goals for the PowerPC 603 microprocessor were to deliver high-performance RISC computing with power consumption low enough for portable laptop computing. Burgess *et. al.* describe features of the PowerPC 603 implementation; while Suessmith and Paap focus their attention to details of the power management features of the PowerPC 603 microprocessor. Determining the impact of implementation choices on performance is an important element of modern processor design. Poursepanj describes the methodology for performance modeling used by Somerset PowerPC designers.

The needs of computer systems developers do not end with the processor silicon. High-quality compilers are equally important to the creation of high-performance systems. Shipnes and Phillip describe the modular approach to the organization of Motorola's optimized PowerPC compilers. Increasingly important is the need to begin software development and testing under software simulation before system hardware is available, and Anderson provides an overview of Motorola's PowerPC simulator family.

The alliance companies, together and with their other partners, continue to develop the other elements needed in a complete PowerPC computer system. A preliminary version of the PowerPC Reference Platform Specification has been distributed to computer system developers. Operating systems such as AIX, Macintosh System 7, Windows NT, Workplace OS, and others are already or will become available for PowerPC systems; as well as chipsets and peripheral devices from IBM, Motorola, and other silicon providers.

A great deal of effort lies behind the creation of this special issue. Special thanks goes to Myhong Nguyenphu of IBM for her immense help in organizing and coordinating the IBM contributions to this issue. We also wish to thank the members of the IBM and Motorola technical community who contributed their time and energy to write and review the articles that follow, as well as the ACM article reviewers who provided many useful suggestions.



PowerPC 603

The Making of the PowerPC



MOTOROLA

Information Systems

Centre for
ing in the Institute at

on the design of
based on the Set
s over the relational
between high-level lan-
es, and lending itself
ising of information.

able in October 1994

la, Pierangela
Mariagrazia Fugini,

ity issues in database
ercial or future sys-
with integrity and con-
of alternative security
cluding those for
ted, frame-based

opportunity to order
made available to the
ntroductory Rate...

Postal Code _____
_____ Exp Date _____
_____ Exp Date _____
_____ Exp Date _____

European Service Center
Marcel Thiry 204
Brussels, Belgium
Phone: 32 2 774 9602
Fax: 32 2 774 9690

ACM_EUROPE@ACM.org

CACM3B94

Charles R. Moore and Russell C. Stanphill

The Making of the PowerPC



The PowerPC Alliance

In September 1991, IBM, Apple and Motorola announced a broad alliance to jointly pursue the development of some of the following emerging technologies. The alliance included five principle initiatives:

- **Object-oriented technology.** Apple and IBM agreed to form Taligent, an independent company, to develop and license a future operating system based on object-oriented design principles.
- **Multimedia technology.** Apple and IBM agreed to form Kaleida, a joint venture company, to create and license multimedia technologies.
- **Interconnectivity and networking.** IBM and Apple agreed to develop hardware and software solutions that allowed their systems to interact more effectively.
- **Open systems environment.** Apple, IBM and Motorola agreed to jointly define the PowerOpen™ environment. PowerOpen supports both AIX™ and Macintosh applications, and will be licensed to other vendors.
- **Microprocessor technology.** IBM and Motorola agreed to jointly develop a broad family of microprocessors based on a derivative of IBM's POWER™ architecture called the PowerPC architecture.™

Central to this alliance is the commitment to the PowerPC Architecture and a family of microprocessors. Es-

entially, the companies have agreed to a vision that brings the architecture to the forefront of the computing world, and enables a broad range of PowerPC microprocessor-based systems. This article will primarily focus on the PowerPC microprocessor technology aspects of the alliance.

PowerPC Microprocessors

In addition to a commitment to the PowerPC Architecture, the alliance also calls for the development of a family of single-chip PowerPC microprocessor implementations. This initial family of devices is charged with providing superior solutions for system designers across a very wide range of system design points. As a result, the microprocessors must be general-purpose and offer versatile interfaces. The economics of the computer industry today demand that these designs be relatively inexpensive and allow for inexpensive system-level solutions. In addition, although the various members of the family may require specific optimization trade-offs for particular market segments, they must all exploit the inherent advantages of the PowerPC

Architecture to achieve high performance.

To develop these devices, IBM and Motorola have jointly formed a PowerPC design center in Austin called Somerset. This jointly managed center is staffed with approximately 300 experienced microprocessor designers. A design methodology has been established that allows the team to achieve high-quality, aggressive designs on relatively short development cycles. This is achieved by blending tools that enhance designer productivity with tools that allow for full custom design and analysis. The tool set allows the designs to be fabricated by either IBM or Motorola manufacturing facilities using a common half-micron CMOS process.

PowerPC Architecture

The PowerPC Architecture is a third-generation RISC architecture that has been optimized for the diverse computing requirements of the future [3]. It has been jointly adopted by IBM, Motorola and Apple. The collective experience of these companies provided a firm foundation for making appropriate architectural

The partners agreed to an initial microprocessor roadmap that specifies four independent design points.

Each design is challenged with bringing industry leadership solutions to their specific segments of the computing market.

trade-offs. The powerful new architecture embraces fundamental concepts of simplicity and general applicability while extending itself to advanced techniques that will carry it into the next decade and beyond.

There were several goals for the PowerPC Architecture. First, it was important that the architecture maintain an application binary interface (ABI) compatible with IBM's POWER architecture. This allows PowerPC microprocessor-based machines to leverage the application base that exists for IBM's RISC System/6000™ machines. To this end, the user-level instruction set and programming model of POWER was selected as a starting point for the PowerPC Architecture. Although some instructions were ultimately added and others deleted, these changes can effectively be managed by compilers and operating systems.

The second goal was to simplify the architecture and ease unnecessary implementation requirements. This flexibility allows implementers to make optimizations that are appropriate for specific market targets. In addition, the simplifications allow for smaller chip sizes, faster cycle times, and more aggressive superscalar implementations.

A third objective of the PowerPC Architecture was to provide support for a wide range of uniprocessor and multiprocessor system configurations. This objective was achieved by recognizing key abstractions of the storage hierarchy and defining the storage control architecture to allow effective management of these abstractions. Furthermore, the architec-

ture allows storage references to follow either a big-endian or a little-endian byte ordering convention.

Finally, the PowerPC Architecture was defined with a set of 64-bit extensions that allow for upward compatibility of 32-bit applications. This was achieved by defining the 64-bit instruction operation as a logical extension to the 32-bit execution model. The memory management architecture was also extended to allow address translation of 64-bit addresses. To allow flexibility, each implementation can be compliant with either the base 32-bit PowerPC Architecture or the extended 64-bit architecture.

The alliance partners agreed to an initial microprocessor roadmap (see Figure 1) that specifies four independent design points. Each of these designs are challenged with bringing industry leadership solutions to their specific segments of the computing market. The initial four design points include the following:

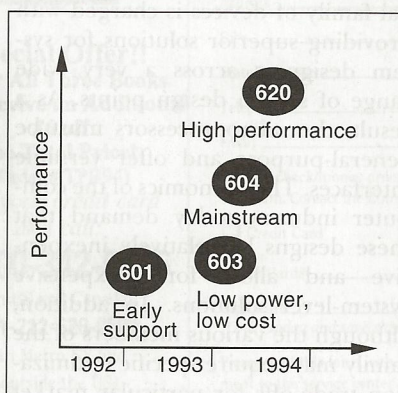


Figure 1. PowerPC roadmap

The **PowerPC 601™ microprocessor** is the first member of the family and is responsible for bringing PowerPC Architecture to the market as early as possible [2]. It is targeted for early PowerPC adopters and is suited for use in desktop computers, portable systems, and low-end multiprocessor systems. The design implements the 32-bit PowerPC Architecture and achieves competitive performance at a relatively low cost. Processor chips have been available since early 1993, and first PowerPC 601 microprocessor-based systems were introduced by IBM in the fall of 1993. The PowerPC 601 is available in 50-, 66-, 80-, and 100MHz versions.

The **PowerPC 603™ processor** is defined to address the low-end and low-cost range of the 32-bit PowerPC microprocessor family [1]. This part packs desktop performance into an 85 square millimeter die, and dissipates less than 3 watts of power at 80MHz. The chip is intended for use in low-end desktop systems and portable systems. This part was announced formally in October 1993, and is slated for production in 3Q94.

The **PowerPC 604™ microprocessor** is designed for mainstream computing environments including personal computers, midrange workstations, and multiprocessor systems. It is organized to offer superior performance over the competition, and outstanding price/performance. The chip is a 32-bit implementation of the PowerPC Architecture, and uses advanced superscalar design techniques to achieve high performance. Samples are expected in the third quarter of 1994.

The **PowerPC 620™** processor is designed to deliver the maximum performance achievable with the currently available half-micron CMOS process technology. This superscalar design implements the full 64-bit PowerPC Architecture and includes an embedded L2 cache controller that interfaces to standard static random access memory (SRAM) chips. The design is targeted at high-end desktop computers, work-group servers and transaction processing-based systems. Samples are expected in the second half of 1994.

These four microprocessors will be heavily used in future IBM and Apple products, and they will also be independently marketed and sold by both IBM and Motorola for use by other system developers. The extension of the PowerPC microprocessor family to the open market is an important and strategic part of the alliance's plan to make PowerPC Architecture successful.

Future Direction

The four microprocessor designs identified on the current roadmap represent only the start of the PowerPC processor development. As technology continues to advance, new opportunities for higher performance, greater levels of integration, lower power consumption and lower cost will arise. IBM and Motorola are committed to driving these technology advancements into their processor development efforts, and to extend the product roadmap to eventually cover an even greater range of computing requirements.

In addition, over time, the existing design points will become usable as cores for more fully integrated and optimized solutions for particular target applications or markets. These cores will also be available for customers to use in development of their own application-specific VLSI solutions. Finally, relevant aspects of the PowerPC Architecture will be driven into other market segment opportunities. For example, in a separate effort, IBM is developing a line of embedded controllers based on portions of the PowerPC Architecture. These controllers will be suitable for environments that are less demanding than the general-purpose computing

environment, yet still require high performance at a very low cost. ■

References

1. Kahle, J. and Ogden, D. The PowerPC 603 microprocessor. *IBM RISC System/6000 Technology: Vol. II*, IBM Corp., 1993.
2. Moore, C.R., The PowerPC 601 microprocessor. *IBM RISC System/6000 Technology: Vol. II*, IBM Corp., 1993.
3. Silha, E. PowerPC Architecture: A high-performance architecture with a history. *IBM RISC System/6000 Technology: Vol. II*, IBM Corp. 1993.

In this document, the terms PowerPC 601 microprocessor and 601; PowerPC 603 microprocessor and 603; PowerPC 604 microprocessor and 604; and PowerPC 620 microprocessor and 620 are used to denote the various microprocessors from the PowerPC Architecture family.

Motorola is a trademark of Motorola, Incorporated.

Apple is a trademark of Apple Computer Corporation.

IBM, PowerPC, PowerPC Architecture, PowerPC 601, PowerPC 603, PowerPC 604, PowerPC 620, POWER, PowerOpen, AIX and POWER Architecture are trademarks of International Business Machines Corporation.

About the Authors:

CHARLES R. MOORE is senior engineer in the RISC System/6000 division of IBM Corp. He is currently working in the high-end processor development group. His research interests include high-performance computer organizations, VLSI design methods and strategic planning. **Author's Present Address:** IBM Corp., 11400 Burnet Rd, M/S 4305, Austin, TX 78758-3493; email: cmoore@ibm.com

RUSSELL C. STANPHILL is a Motorola veteran, currently serving as codirector of the Somerset Design Center. In 1991, Stanphill was instrumental in working with Apple and IBM to form what is now the PowerPC alliance. By early 1992 he was elected to manage the resulting Somerset facilities. **Author's Present Address:** Motorola, Semiconductor Products Sector, Microprocessor and Memory Technologies Group, 6501 William Cannon Drive West, Austin, TX 78735-8598.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/94/0300 \$3.50

Big CASE tool vendors caught with their pants down!

"Object modeling and C++ programming, side-by-side, always up-to-date."

What if you could have your OOA/OOD model and all of your C++ code continuously up-to-date, all the time, throughout your development effort?

Consider the possibilities...

In one window, you see an object model, with automatic, semi-automatic, and manual layout modes, plus complete view management. Side-by-side, in another window, you see fully-parsed C++ code. You edit in one window or the other. Press a key. Both windows agree with each other. **Together.**

Or suppose that you are working on a project with some existing code. (That's no surprise; who'd consider developing in C++ without some off-the-shelf classes?) You read the code in. Hit a button. And seconds later, you see an object model, automatically laid out for you, ready for you to study side-by-side with the C++ code itself. **Together.**

Or suppose you are building software with other people (that's no surprise either). You collaborate with others and develop software with a lot less hassle, because the fully integrated configuration management features help you keep it all...**Together.**

The name of this product? It's earned the name...

Together/C++
continuously up-to-date
object modeling and C++ programming

Key features. Continuously up-to-date object modeling and C++ programming, side-by-side, so you can work back-and-forth between the two (and let the tool keep them in-sync).

Automatic, semi-automatic, and manual layout of object models, so you can feed in existing class libraries and quickly see a meaningful object model.

Object modeling view management, including view control by C++ construct, regular expression, proximity, layer, or directory (you control model complexity at all times).

Fully flexible documentation generation, version control, and SQL generation, too.

"State-of-the-art application development."

-- Computerworld/Germany

"You've really hit the nail on the head when it comes to reverse engineering existing C++ code. No other tool comes close to the power and capability of Together/C++."

-- Russell Rudduck, Perot Systems

Money-back guarantee. Purchase Together/C++ and try it out risk-free. You get a full 30-day money-back guarantee.

How to order. Order Together/C++ by purchase order, check, or credit card. To order, or for more information, please call 1-800-OOA-2-OOP (1-800-662-2667, 24 hours, 7 days a week). Or contact:

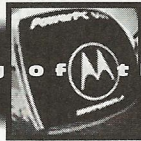
Object International, Inc.
Education - Tools - Consulting
8140 N. MoPac 4-200
Austin TX 78759 USA
1-512-795-0202 - fax 795-0332
e-mail: object@acm.org

Outside of North America, contact:
Object Int'l Ltd.
Eduard-Pfeiffer-Str. 73
D-70192 Stuttgart, Germany
++49-711-225-740 fax ++49-711-299-1032
e-mail: 100034.1370@compuserve.com

©1994 Object Int'l, Inc.
All rights reserved.
"Together" is a trademark of Object Int'l, Inc.
CACM694

In Great Britain and Ireland, contact:
Object UK Ltd.
++44-703-39-390 fax ++44-703-399-991
e-mail: 100024.2720@compuserve.com

The Making of the PowerPC



History of the *PowerPC Architecture*

Approximately three years ago Apple, IBM, and Motorola decided to adopt a common RISC architecture on which to base their future hardware and software systems. The three companies believed the technical advantages of RISC were sufficiently compelling that if significant resources could be concentrated on one architecture, it might be possible to leverage these advantages into a serious market contender. Thus was born the PowerPC Architecture.

Although confident of the significant advantages of RISC, they also realized that market success ultimately depends on software. To get a running start, it seemed sensible to begin with a RISC architecture that already had a large installed base of software. From that perspective, the obvious choice was IBM's successful line of RISC System/6000 workstations and servers, which are based on the company's POWER architecture.

The PowerPC Architecture reflects the work done by the team of computer architects from Apple, IBM, and Motorola who were tasked with the objective of retargeting the architecture into a form more suitable for high-volume, single-chip microprocessors. The architects also enhanced the architecture with better multiprocessor support features and extended it with a 64-bit address capability in order to ensure its viability into the next century. This article describes the evolution to the PowerPC Archi-

ture and briefly describes some of the changes incorporated relative to the POWER architecture.

The fundamental concepts of RISC were developed by John Cocke in the mid-1970s at IBM's T.J. Watson Research Center, and first embodied in a machine called the IBM 801 minicomputer [5]. These ideas were further refined and articulated by a group at the University of California at Berkeley led by David Patterson, who coined the term "RISC" [4]. These early pioneers realized that RISC represented a substantive departure from the then-popular trend toward more complex instruction sets (embraced by "CISC" architectures such as the VAX, 8086, 32000, and 68000) that promised higher performance, less cost, and faster design time.

Complex instruction set architectures were primarily motivated by a desire to reduce the "semantic gap" between the machine language of the

processor and the high-level languages in which people were programming. The theory was that such a processor would have to execute fewer instructions (have a shorter path length) and would, therefore, have better performance. The key observation underlying RISC, however, was that the sequential microcode interpreter required to execute these complex instructions introduced an expensive overhead that actually slowed down execution of the more frequently occurring simple instructions—resulting in a net *loss* in performance. Furthermore, complex instructions proved to be a rather poor target for compilers that had difficulty using them and in many cases their use precluded optimizing out unnecessary operations.

With the declining cost of memory devices and improved compiler technology, it became feasible to consider simplifying the instruction set, even at the cost of larger code size and

higher memory bandwidth requirements. The 801 was the first machine to implement this strategy. It successfully demonstrated that simplifying the instruction set enabled implementations with smoother running (bubble-free) pipelines that could approach the goal of single-cycle instruction throughput. It was also discovered that other architectural features now associated with RISC—such as a large uniform register file—enabled compiler optimizations that actually kept code expansion very low and even reduced data bandwidth relative to existing CISC architectures. On balance, the 801 demonstrated that investing more transistors in instruction throughput, fast cycle times, and more registers, produced a better solution to the computer performance equation than was possible by spending those transistors on more complex instructions.

But the ideas behind RISC turned out to be even more significant than originally thought. Not only did RISC processors demonstrate more parallelism through better pipelining, but the resulting simplification of the hardware made tractable the idea of dispatching multiple instructions simultaneously (superscalar¹) and enabled implementation of the heretofore mainframe-domain concepts of dynamic instruction reordering and out-of-order instruction execution on single-chip microprocessors. This is the essence of RISC architecture—it allows the execution of more operations in parallel and at a higher rate than possible with a CISC architecture employing similar implementation complexity.

Satisfied that the 801 concepts had made significant improvements in instruction cycle times and pipeline efficiency, IBM set out to improve further on the 801 architecture by: 1) explicitly embodying the concept of superscalar operation in the architecture; 2) improving the architecture as a target for compilers; 3) reducing instruction path lengths; and 4) including floating point as a first-class data type in the architecture.

¹ The term "superscalar" is believed to have been coined by T. Agerwala and John Cocke [1]. It refers to machines capable of dispatching multiple instructions per clock from a conventional linear instruction stream.

This effort culminated in the development of the POWER architecture [3] in the late 1980s, which now forms the basis of IBM's RISC System/6000 family of workstations and servers (see Figure 1).

The POWER Architecture

The POWER architecture is a conventional RISC architecture in most respects; it adheres to the most important RISC tenants of fixed-length instructions, register-to-register architecture, simple addressing modes,

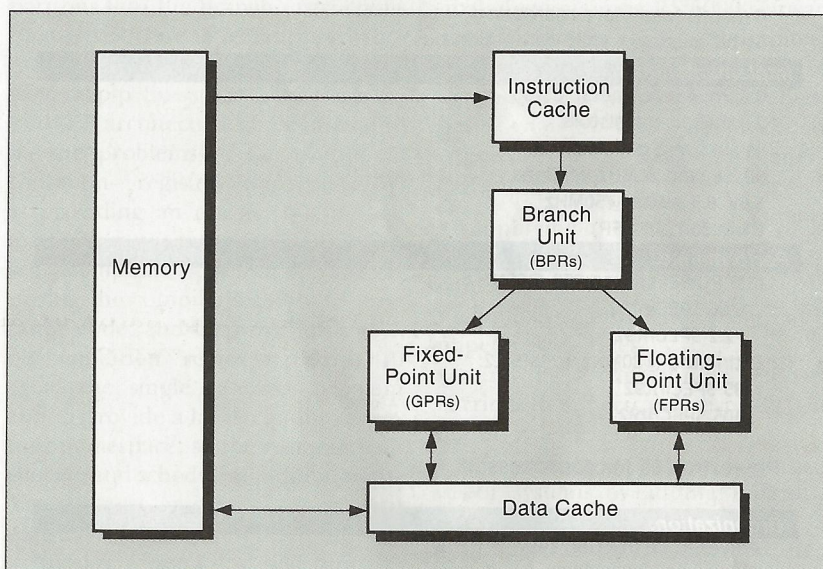
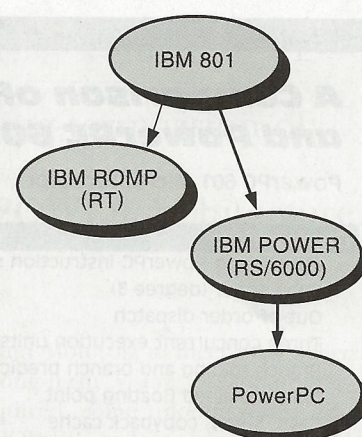


Figure 1. PowerPC genealogy

Figure 2. POWER Architecture model

simple (not requiring microcode interpretation) instructions, a large register file, and a three-operand (non-destructive) instruction format. However, the POWER architecture also has several additional features that set it apart from other RISC architectures.

First, the instruction set was organized around the idea of superscalar instruction dispatch. Conceptually, instructions are dispatched across three independent execution units, a branch unit, a fixed-point unit, and a floating-point unit (see Figure 2). Instructions can be dispatched to each of these units simultaneously where they can execute concurrently and finish out of order. To increase the level of instruction parallelism that can be achieved in practice, the instruction set architecture defines an independent set of register resources for each unit. This minimizes the

communication and synchronization required between units, thus allowing execution units to adjust to the dynamic instruction mix by "slipping" past one another. Any data communication required between units must be performed explicitly, exposing it to the compiler, where it can be effectively scheduled. (It is important to realize this is a conceptual model only. Any given processor may implement each of the conceptual units as multiple execution units to support additional instruction parallelism. But the existence of the model led to the consistent design of an instruction set that naturally supported at least degree three parallelism.)

Second, the POWER architecture

A Comparison of PowerPC 601 and PowerPC 603 Features

PowerPC 601 Microprocessor

Organization:

- POWER and PowerPC instruction sets
- Superscalar (degree 3)
- Out-of-order dispatch
- Three concurrent execution units
- Branch folding and branch prediction
- Fully pipelined floating point
- 32KB, 8-way, copyback cache
- MESI cache coherency
- Dynamic load-store reordering
- 64-bit, bursting, split-transaction bus

Implementation:

- 0.6 μ m, 4-level metal, CMOS
- 2.8 million transistors
- 11 $\times$ 11 mm die size
- 50, 66, and 80MHz versions
- 3.6v, 6.5 watts @50MHz
- C4 in 304 pin QFP

Performance:

- IBM POWERstation 250 @66MHz:
 - 62.6 SPECint92
 - 72.2 SPECflt92
- Estimated @80MHz w/1MB L2:
 - 85 SPECint92
 - 105 SPECflt92

PowerPC 603 Microprocessor

Organization:

- PowerPC instruction sets
- Superscalar (degree 3)
- Out-of-order execution and completion
- Five concurrent execution units
- Register renaming
- Reservation stations
- Branch prediction and speculative execution
- Fully pipelined floating point
- 8KB/8KB, 2-way, caches
- Dynamic load-store reordering
- 64-bit, bursting, split-transaction bus

Implementation:

- 0.5 μ m, 4-level metal, CMOS
- 1.6 million transistors
- 7.4 $\times$ 11.5mm die size
- 66, and 80MHz versions
- 3.3v, 3 watts @80MHz
- 240 pin CQFP

Performance:

- Estimated @80MHz w/1MB L2:
 - 75 SPECint92
 - 85 SPECflt92

added several "compound" instructions to reduce instruction path lengths. Perhaps the *only* drawback to RISC technology relative to CISC is that it sometimes takes more instructions to perform a given task. IBM discovered that most of this code expansion is avoidable with minor enhancements to the instruction set that do not constitute a return to CISC-like complex instructions. For example, a large fraction of the code expansion was found in the prolog and epilog code associated with saving and restoring registers across a procedure call. To eliminate this as a factor, IBM introduced "load-and-store multiple" instructions that allow several registers to be moved to or from memory with a single instruction. The linkage conventions used by the POWER compilers addressed the problems of relocation, shared libraries, and dynamic linkage in one simple, unified mechanism. This is done by indirect addressing through a table of contents (TOC) that is updated at load time. The load-and-store multiple instructions were important to these linkage conventions.

Another example of "compound" instructions is the optional update of the base register on loads and stores with the newly calculated effective address. This instruction eliminates the need for the extra **add** instruction that would otherwise be required to increment the index for progressive indexing of arrays. Even though this is a compound operation, it does not adversely affect the conventional RISC pipeline flow because the updated address is already computed and a register file write port is normally available while waiting on the memory operation.

The POWER architecture provided a few other path length reducing instructions such as: an extensive set of bit-field manipulation instructions, compound multiply-add floating-point instructions, condition register setting as a side effect of normal instruction execution; and load and store string instructions (which load or store arbitrarily aligned byte strings).

A third factor that differentiates the POWER architecture from many other RISC architectures is the absence of the branch-and-execute ca-

Satisfying the diverse needs of three major corporations like Apple, IBM, and Motorola to meet their collective long-term vision of computing required some modifications to the POWER architecture.

ability. Branch-and-execute (sometimes called delayed branching) causes the instruction following a branch to execute before the branch gets taken. This feature worked effectively in early RISC machines to fill the instruction bubble created by branch evaluation and fetching the new instruction stream. However, in more advanced, superscalar machines, this feature is ineffective because a single branch delay cycle induces multiple instruction bubbles that cannot all be covered with a single architectural delay slot. Almost all such machines will implement exotic facilities (e.g., branch target caches) for covering these bubbles. These facilities render the delayed branch useless. Not only is the delayed branch ineffective in such machines, it introduces significant complexity into the instruction sequencing logic. Thus, even though the 801 employed branch-and-execute, it was not included in the POWER architecture. Instead the POWER branch architecture was organized to support branch-lookahead and branch-folding techniques as described in the next paragraph.

The branching technique used in the POWER architecture is the fourth unique feature of the architecture compared to other RISC processors. The POWER architecture defines an enhanced condition register facility. The problem with traditional condition register architectures is that the setting of condition bits as a side effect of instruction execution poses serious limitations on the compiler's ability to reschedule instructions. Additionally, a condition register represents a single architectural resource that causes a serious bottleneck in a machine that executes multiple instructions in parallel or out of order. Some RISC architectures avoid the problem by completely eliminating

the condition register and requiring conditions to be explicitly set (by a compare instruction) in a general register and/or by folding the comparisons into the branch instructions themselves. The latter approach potentially overloads the branch-execute pipeline stage. Therefore the POWER architecture chose instead to fix the problems of the traditional condition register approach by: a) providing an opcode bit in each instruction to make the condition register update optional, thereby restoring the compiler's ability to rearrange code, and b) providing multiple condition registers (eight) to avoid the single resource problem and to provide a large condition register namespace, so the compiler can allocate and schedule condition register resources as it does for general registers.

Another reason for selecting the enhanced condition register model was that it is consistent with the organization of the machine into independent execution units. Conceptually, the condition register is local to the branch unit. Consequently, it is not necessary to access the general register file (which is local to the fixed-point unit) to evaluate and execute a conditional branch. To the extent the compiler can schedule condition code updates early (and/or load the branch address registers early) the hardware can lookahead and fold-out resolvable branches from the instruction stream. This avoids the instruction issue slot normally occupied by the branch instruction, and allows the instruction dispatcher to feed a continuous linear stream of instructions to the computational execution units.

Evolution of the PowerPC Architecture

Satisfying the diverse needs of three major corporations like Apple, IBM,

and Motorola to meet their collective long-term vision of computing required some modifications to the POWER architecture. So, with the goal of maintaining RS/6000 software compatibility, a team of architects from IBM, Apple, and Motorola set out to refine the architecture. A number of changes were made to the architecture in the following general categories:

- simplifying the architecture to be more appropriate for low-cost single-chip microprocessors
- eliminating instructions that might impede clock rates
- removing architecturally imposed barriers to superscalar dispatch and out-of-order execution
- encouraging symmetric multiprocessor systems by adding multiprocessor support features
- adding new features deemed necessary for anticipated applications
- clearly defining the line between "architecture" and "implementation"
- assuring a long lifetime for the architecture by extending it to a true 64-bit architecture

These changes resulted in a new architecture, officially called the PowerPC Architecture, which will form the basis for next-generation products not only from the three founding companies but from a large number of other companies as well.

The PowerPC Architecture maintains the same basic programming model and instruction opcode assignments as the POWER architecture. Where changes were made that could potentially prevent PowerPC processors from running existing RS/6000 binaries, care was taken to remove or change the feature in such a way that it could be trapped and emulated in software. To make this approach practical, features were changed only if they were either used infrequently

in application code or were isolated in library routines that could easily be replaced.

Some of the more significant changes made in going from the POWER to the PowerPC architectures in the categories listed previously include the following:

- *Simplification*
 - eliminated several bit-field instructions that used three source operands (to avoid the need for an extra general register file port)
 - redefined the real-time-clock as a simple binary counter with variable count rate
 - eliminated special segments and the associated fine-grain locking
 - eliminated the most complex string instruction
- *Higher clock rates*
 - eliminated four instructions whose operation was dependent on the value of the source operand
- *Removing superscalar barriers*
 - eliminated the MQ register and all extended precision shifts, extended integer multiply, and the divide-with-remainder instructions that used it
 - added subtract without carry
 - added floating-point imprecise exception modes
- *Multiprocessor support*
 - added reservation model for atomically updating shared memory
 - defined the weakly ordered storage model
 - added new memory transaction ordering instructions
 - redefined the user mode cache control instructions for use in multiprocessor systems
 - defined memory aliasing rules
 - replaced two-level inverted page table with single hashed page table capable of supporting concurrent aliasing
- *New features*
 - added single-precision floating-point instructions
 - added unsigned fixed-point multiply and divide
 - added new storage attribute controls on memory pages
 - added little-endian memory addressing mode

- added variable-size block address translation capability
- *Extension to 64 bits*
 - defined superset architecture that supports full 64-bit linear logical address space and 64-bit integer arithmetic
 - defined segment table to provide an 80-bit virtual address space (to replace the segment registers used in 32-bit addressing that provided a 52-bit virtual address space)
 - extended the page table formats to support a full 64-bit physical address space

The most far-reaching change to the architecture was the extension to 64 bits, which involved a number of modifications to the user programming model, instruction set, and address translation mechanisms. We defined the PowerPC Architecture as a full 64-bit architecture which has a 32-bit subset. The architecture permits both 32- and 64-bit versions of PowerPC processors, but all processors are required to support 32-bit programs as a minimum. The architecture defines a 32/64-bit mode switch controllable from supervisor code that allows a 64-bit processor implementation to run 32-bit programs.

The primary change to the user-visible architecture was to extend the width of the general registers and the branch address register to 64 bits. On processors that implement the full 64-bit architecture, all instructions now simply work on full 64-bit registers rather than 32-bit wide registers. Nearly all instructions are entirely mode-independent. The only significant effect of the mode switch is to select how much of a 64-bit effective address is used in address translation. (There are a few other minor effects such as selecting the ALU bit from which ALU conditions such as carry and overflow are generated.) The address translation mechanism for 64-bit processors is similar to the translation mechanism used on 32-bit implementations, except that the segment registers are replaced with a segment table to handle the larger logical and virtual address spaces, and the page table format was extended to accommodate the larger

virtual and physical address spaces. But these translation changes are not apparent to application software.

The PowerPC 601

The first PowerPC microprocessor, the PowerPC 601 microprocessor, is now available from both IBM and Motorola. The 601 is a medium-sized, medium-performance processor suitable for low- to medium-cost desktop computer systems. It was designed as a transition processor from the POWER architecture to the PowerPC Architecture. Thus it implements a superset of both POWER and PowerPC features so that existing RS/6000 binaries will run at full speed. This provides additional time for the compilers to be retargeted to the PowerPC Architecture and applications to be recompiled before the processors which implement strictly the PowerPC Architecture become widely available.

The 601 was based on an IBM single-chip processor that was being designed when the alliance was first formed. But the 601 underwent major enhancements to improve performance and reduce costs (see sidebar "A Comparison of PowerPC 601 and PowerPC 603 Features.") For example, a more sophisticated branch unit, enhanced with multiprocessor features including Motorola's 88110 high-performance microprocessor bus interface was included. The 601 implements a moderately aggressive superscalar microarchitecture capable of dispatching 3 instructions, possibly out-of-order, on each clock cycle.

Other PowerPC Processors

IBM and Motorola, with Apple engineering participation, have put into operation a new design center to develop future PowerPC microprocessors. The Somerset Design Center is a 37,000 square-foot facility located in Austin, Texas, staffed primarily by Motorola and IBM with approximately 300 engineering professionals. The design center is presently working concurrently on three separate PowerPC microprocessors. The three parts currently in development in the design center include:

- The 603: a processor intended pri-

marily for the cost-sensitive, desktop and portable personal computer systems

- The 604: a high-performance part for uniprocessor or multiprocessor desktop personal computers and workstations
- The 620: a 64-bit high-performance part for high-end workstations, servers, and multiprocessor systems

Engineers at the Somerset Design Center employ a formal VLSI design methodology derived from the best of both IBM's and Motorola's CAD tools. The new designs all use an advanced 0.5 μ m CMOS technology using a common set of design rules for both IBM and Motorola semiconductor fabrication facilities.

The first processor designed completely in the new design center was the 603, which is now sampling. The 603 employs a slightly more aggressive microarchitecture than the 601, but has a smaller cache giving it approximately the same performance as the 601 at lower cost (see sidebar "A Comparison of PowerPC 601 and PowerPC 603 Features"). The 603 is also superscalar, capable of dispatching 3 instructions per clock, in-order, into 5 concurrent execution units. It employs register renaming, reservation stations, speculative execution, and out-of-order instruction execution and completion to boost instruction parallelism. The 603 operates at 3.3v and utilizes static design, automatic power-down circuitry, and a number of software-controlled power-saving modes to make it useful in laptop systems as well as low-cost desktop systems.

The 604 and 620 products are not yet officially announced during the development time of this article, but are scheduled to see silicon during 1994. In addition to these four processors, other new PowerPC processors are in development at both Motorola and IBM internal design centers. These designs will target a range of specific markets ranging from very low-cost, high-volume embedded control markets, to very low-power subnotebook computer markets, all the way to very high-end computers. Also, research is under way into advanced microarchitectural

techniques for the next generation of billion-instruction-per-second class microprocessors.

Summary

The PowerPC Architecture is the product of nearly 20 years of work on RISC architectures beginning with IBM's seminal 801 minicomputer in the 1970s and finally refined by Apple's experience in advanced personal computers and by Motorola's experience in delivering low-cost single-chip microprocessors into high-volume markets. This architecture is currently being used as the base for a wide variety of instruction-set compatible microprocessors. With their combined resources, IBM, Apple, and Motorola intend to deliver an unparalleled range of PowerPC RISC processors into the market, all the way from the very lowest end of the portable computer marketplace to the very highest end of the supercomputer market. □

References

1. Agerwala, T. and Cocke, J. High performance reduced instruction set processors. IBM Tech. Rep., March 1987.
2. Gullette, B. The design of the 88110 Bus Interface. In *Proceedings of RISC '92*, Feb. 1992.

3. Oehler, R.R. and Groves, R.D. IBM RISC System/6000 processor architecture. *IBM J. Res. Develop.* 34 (Jan. 1990), 23-36.
4. Patterson, D.S. and Ditzel, D.R. The case for the reduced instruction set computer. *Comput. Architecture News* (Oct. 15, 1980).
5. Radin, G. The 801 Minicomputer. In *Proceedings of the Symposium on Architectural Support for Programming Languages* (March 1982), 39-47.

About the Author:

KEITH DIEFENDORFF is a microprocessor architect and member of the technical staff in Motorola's RISC microprocessor division. Current research interests include advanced architectural concepts for future high-performance microprocessors. **Author's Present Address:** Motorola, Inc., 6501 William Cannon Drive West, Austin, TX 78735-8598; email: keith_diefendorff@email.sps.mot.com

IBM, PowerPC, PowerPC 603, PowerPC Architecture, and RISC System/6000 are trademarks of International Business Machines Corporation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/94/0600 \$3.50

Join The First Society in Computing...

THE ASSOCIATION FOR COMPUTING MACHINERY

Call Today For More Information

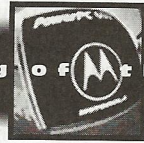
1-800-342-6626

(In U.S. & Canada)

1-212-626-0500

(In Metro N.Y. & Outside U.S.)

The Making of the PowerPC



The PowerPC 603 Microprocessor

In October 1993, Motorola and IBM unveiled the first low-power version of the PowerPC family—the PowerPC 603 microprocessor. Measuring a mere 85mm^2 ($7.4 \times 11.5 \text{ mm}$) in size, the 603 contains 1.6 million transistors and consumes less than 3 watts of power when operating at 80MHz. With estimated performance values of 75 SPECInt92 and 85 SPECfp92, the 603 is comparable in performance to present-day high-end personal computer and workstation processors. This member of the PowerPC family is designed to bring high-performance and low-power capabilities to the laptop and low-cost desktop computer markets.

Following closely on the heels of its predecessor, the PowerPC 601 microprocessor [1], the 603 microprocessor was developed at the joint Motorola/IBM/Apple Somerset Design Center in Austin, Texas. The 603 microarchitecture evolved from Apple, IBM, and Motorola's collective experience on several past designs. The similarity of the POWER and PowerPC architectures permitted the use of sample traces generated by RISC System/6000 machines for evaluation of design trade-offs. The compiler groups also provided their insight to ensure the traces from the past generation of processors and compilers, with their own specific peculiarities, did not misguide the 603's microarchitecture definition, and that trade-offs selected were appropriate for the next generation of compilers.

To accelerate the design and test process, engineers employed a formal

VLSI design methodology derived from the best of both IBM and Motorola's CAD tools. These tools enable both the rapid design and dense packing capability necessary to produce very high-volume, high-yield microprocessors for the commercial market. The 603 design team employed a combination of custom circuitry (for arrays), library components (for data paths), and standard cell place and route (for random logic) to accomplish the 603 design.

Using the best tools and methodology available, the design team took the 603 from concept to working silicon in 18 months. Ongoing design evaluation and debugging, including simulation of 28 billion processor cycles prior to tape-out, provided fully functional first-pass silicon that ran at the design target speed of 80MHz.

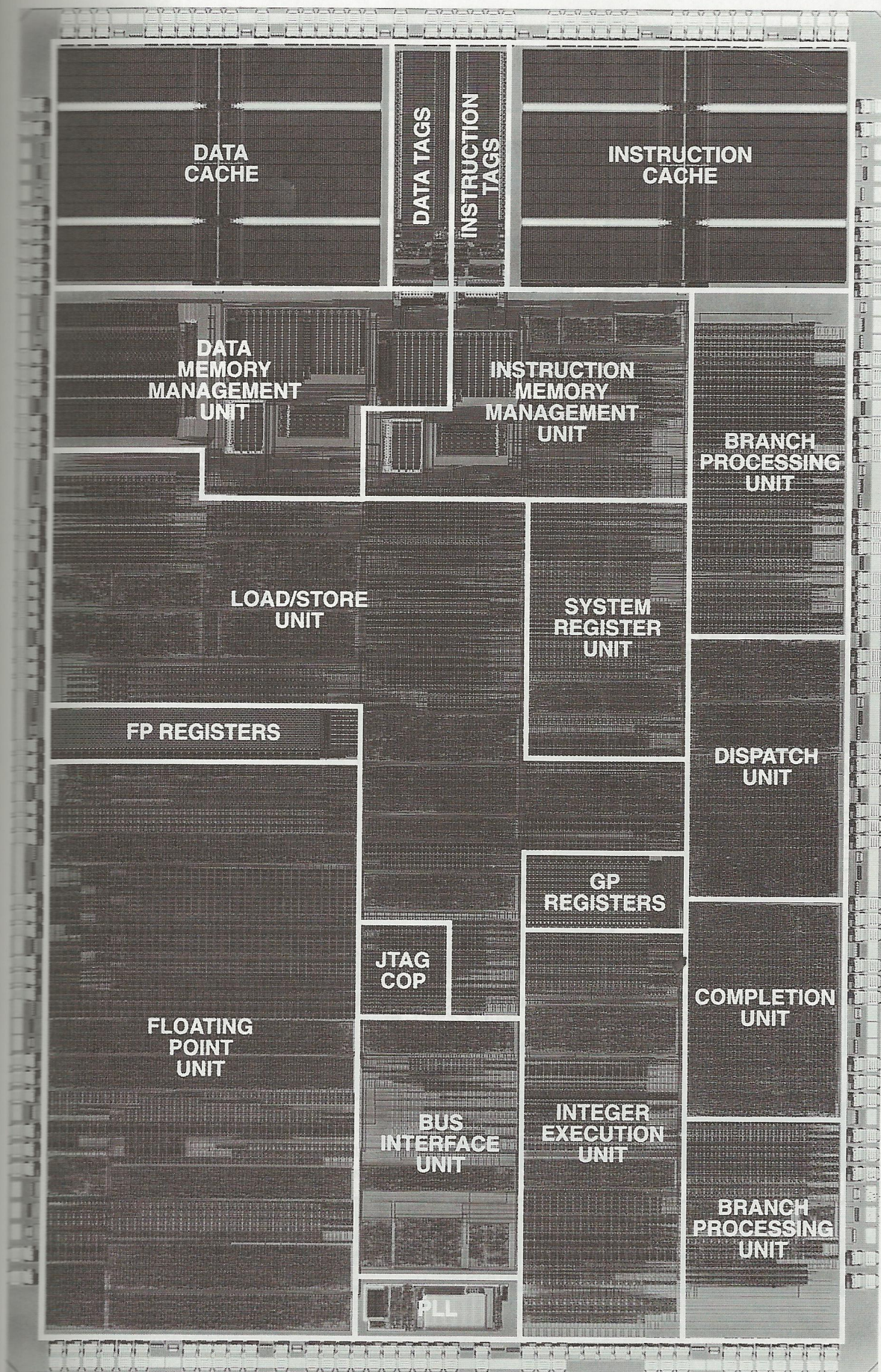
The PowerPC 603 microprocessor is manufactured by Motorola in Aus-

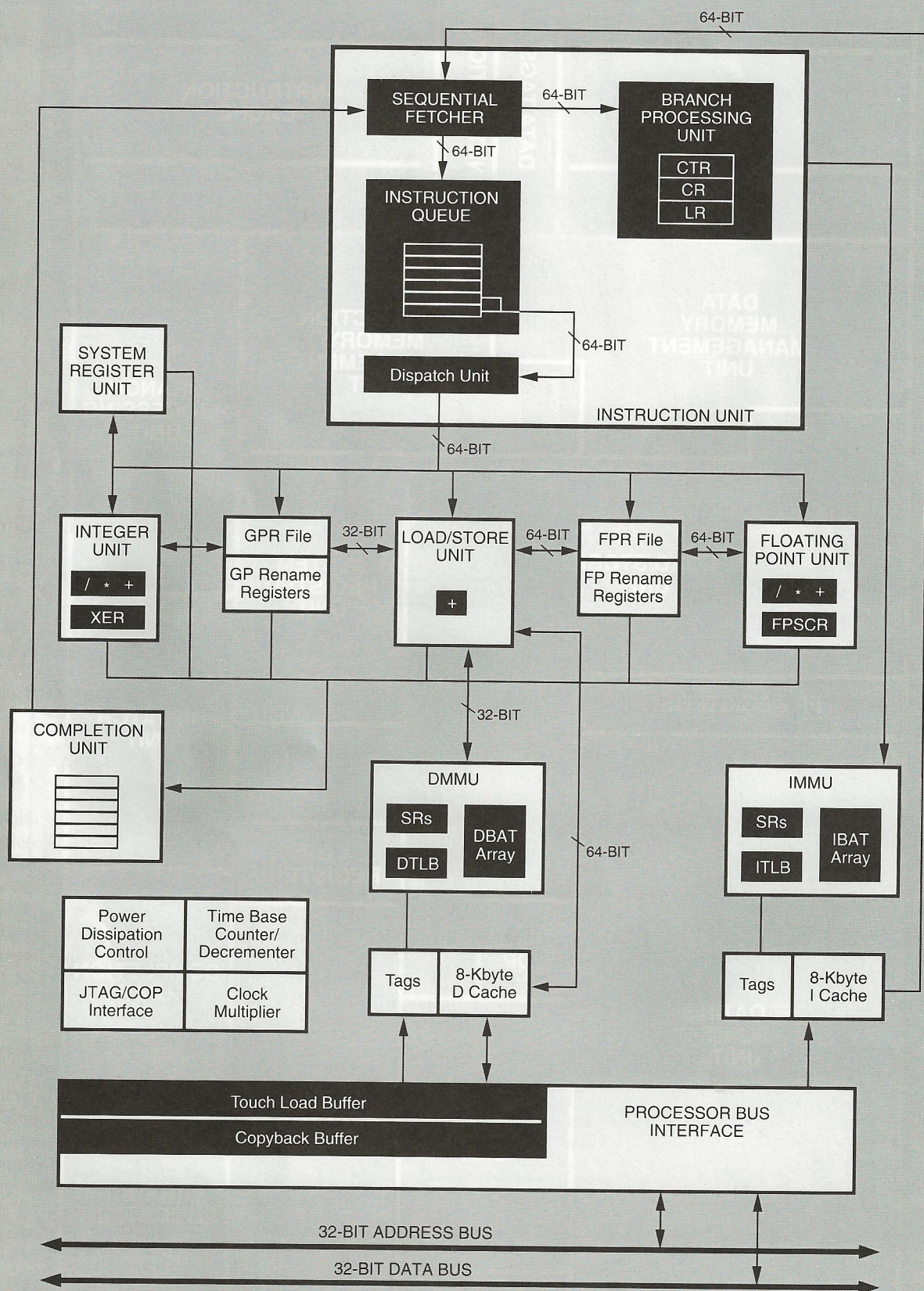
tin, Texas, and by IBM in Burlington, Vt. Motorola and IBM both fabricate the 603 using a $0.5\mu\text{m}$, 4-level metal, 3.3VDC CMOS process with design rules compatible with both companies' semiconductor processes. The die is designed to be packaged in either a 240-pin ceramic quad flat pack or a ball-grid array package. Figure 1 is a photograph of the 603 die.

Functional Overview

The 603 is the first processor in the PowerPC family to fully support the PowerPC Architecture. It incorporates five execution units: branch, integer, floating-point, load/store, and system register; and a pair of on-chip 8KB instruction and data caches.

Figure 1. PowerPC 603 microprocessor die photograph





Since the 603 is a super-scalar microprocessor, it is capable of issuing and retiring as many as three instructions per clock to these execution units. For increased performance, the 603 allows instructions to be executed out-of-order. Additionally, the 603 provides programmable power reduction modes that permit systems designers the flexibility of implementing a variety of power management techniques. A block diagram of the 603 is shown in Figure 2.

Instructions are dispatched in order to one of the five execution units. If there are no operand dependencies, execution occurs immediately. The integer unit executes most instructions in one cycle. The floating-point unit is pipelined and executes both single and double precision floating-point operations. Branch resolution is handled by the branch unit. If the branch conditions are available, branches are immediately resolved; otherwise, instruction execution continues speculatively. Instructions that modify the processor control registers are executed by the system register unit. Finally, data movement between the data cache and the general-purpose and floating-point registers is handled by the load/store unit.

In case of cache misses, the caches access main memory through a 64-bit high-performance bus similar to that of the MC88110 [8]. To maximize throughput and thus increase overall performance, the cache communicates with memory mostly via burst operations that allow a cache line to be filled in one transaction.

After an instruction finishes execution in an execution unit, its results are forwarded to a completion buffer, and then subsequently written to the appropriate register file set when the instruction is retired from the completion buffer. To avoid register contention, the 603 provides separate 32-entry integer general-purpose registers (GPRs) and floating-point register (FPRs) sets for the storage of operands.

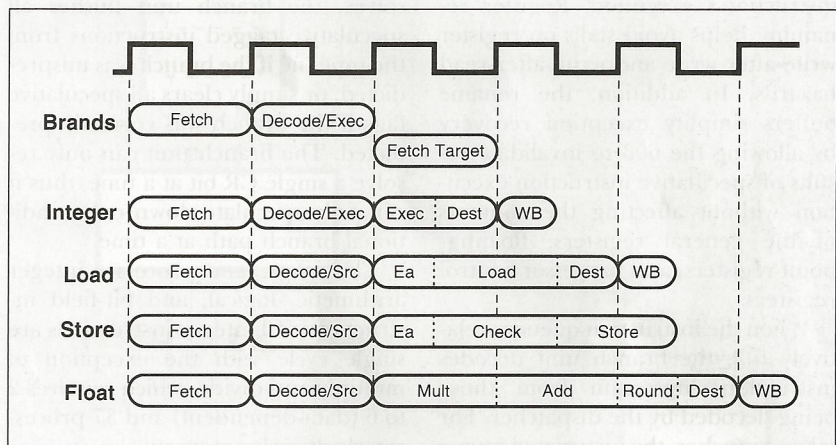
The following sections discuss in

more detail the factors that contribute to the efficient flow of instructions and data through the 603.

Instruction Pipeline

Figure 3 shows the 603's instruction pipelines for several types of instructions.

Fetch stage. During this stage, the instruction fetcher retrieves two instructions at a time from the instruction cache (regardless of alignment), unless the address points to the last word of a cache line, in which case only a single word is returned.



Decode/Source stage. During this stage, the dispatcher and branch unit decode instructions, allocate rename registers, read available source operands, and dispatch instructions to their respective execution units (or reservation stations).

Execute stage. During this stage, the execution units execute instructions and write results back to the destination rename registers. If the data is needed as a source operand for another instruction, the data is forwarded immediately to the requesting unit. When the execution unit finishes with an instruction it notifies the completion buffer that the instruction is finished and tags the instruction if any exceptions occurred.

Completion stage. During this final stage, the completion buffer logic writes the contents of any renamed registers into the architectural registers. It then deallocates rename registers and returns them to the pool for future use. If the completion logic

detects that an instruction is tagged as having caused an exception, it flushes the pipeline and initiates exception processing. Otherwise, it retires completed instructions and removes them from the completion buffer. Because the completion logic retires all instructions in program order, all exceptions are fully precise.

Functional Units

Dispatch unit. The instruction flow diagram is shown in Figure 4. Instructions are fetched two per cycle from the instruction cache, and

Figure 3. PowerPC 603 microprocessor master instruction pipeline

placed in a six-entry instruction queue. On arrival at the instruction queue, branch instructions are immediately forwarded to the branch processing unit for resolution. All other instructions are issued from the instruction queue at the rate of two per cycle if there is no resource contention (for example, a busy execution unit, or a full completion queue).

The instruction dispatcher decodes the bottom two entries of the queue and dispatches up to two instructions per cycle, in program order, to either the integer unit, floating-point unit, load/store unit, or system unit. If the instruction dispatcher finds an execution unit busy, it does not dispatch the instruction and stalls. There are several mechanisms in the 603 to avoid dispatch stalls.

Figure 2. Block diagram of the PowerPC 603 microprocessor

To avoid dispatch stalls due to operand dependencies, the 603 has single-entry zero-latency reservation stations associated with each execution unit. The reservation station holds the instruction until all operands are available. This allows the dispatcher to dispatch subsequent instructions to other execution units without stalling the instruction queue.

In addition to dispatching instructions, the dispatcher allocates rename buffers and coordinates pipeline stalls. Rename buffers provide temporary storage for the results of an instruction's execution. Register renaming helps avoid stalls on register write-after-write and write-after-read hazards. In addition, the rename buffers simplify exception recovery by allowing the 603 to invalidate results of speculative instruction execution without affecting the contents of the general registers, floating-point registers, and processor control registers.

When the instruction queue is relatively full, the branch unit decodes instructions upstream from those being decoded by the dispatcher. For taken branches, the instruction queue usually contains enough instructions to keep the dispatcher busy until the new instruction stream is fetched. Thus, from the dispatcher's perspective this allows basic blocks to be connected without ever seeing the branch instruction or the fetch penalty. Once dispatched, the dispatcher transfers control of the instruction's execution to the execution unit, and control of the overall instruction stream to the completion buffer logic.

Execution units. The *branch unit* decodes the fetched instructions and executes most branch instructions in a single clock. It also retires and eliminates from the instruction queue (branch folding) branches that do not modify branch-related resources such as the Link or Count registers. The Link register contains the return address from a branch instruction. The Count register is a loop counter that is used by some branch instructions.

The branch unit has its own facilities for calculating the branch target address. Conditional branches in the PowerPC Architecture depend on a

counter-register (CTR) value and/or any one of 32 condition register (CR) bits.

If the CTR value is unavailable, the branch unit and instruction fetching is stalled (not likely to occur very often). If a CR bit is unavailable, the branch unit predicts either the taken or fallthrough path based on bits in the branch opcode. These predicted instructions are tagged as speculative and proceed down the pipeline normally. The unavailable CR bit that initiated speculative execution is checked each cycle. When the CR bit becomes available and the branch resolves, the branch unit flushes all speculative tagged instructions from the pipeline if the branch was mispredicted, or simply clears all speculative tags if the branch was correctly predicted. The branch unit can only resolve a single CR bit at a time, thus it can only speculate down one conditional branch path at a time.

The *integer unit* processes integer arithmetic, logical, and bit-field instructions. All integer instructions are single cycle with the exception of multiply and divide, which requires 2 to 6 (data-dependent) and 37 processor clock cycles respectively.

The *load/store unit* handles load and store instructions to and from both the integer and floating-point registers. It contains a dedicated adder for the calculation of effective addresses, and the logic required for data alignment to and from the cache.

The load/store unit is fully pipelined so that loads can be dispatched at the rate of one per clock cycle. Loads have a two-clock-cycle latency, a half cycle to compute the effective address, one cycle to access the data cache and MMU, and another half cycle to write the result into the rename register.

Since the load/store unit cannot write to the cache until after checking for memory protection violations, it does not execute store instructions in a fully pipelined manner. During the execution stage, the load/store unit calculates the effective address and translates the address to check for memory protection violations. On the next clock cycle, the store advances to a holding buffer, where it waits for the completion logic to retire the instruction and enable writing data to

the cache.

The 603 takes advantage of the PowerPC Architecture's weakly ordered memory model and allows load instructions to bypass pending stores in order to minimize stalls due to load data hazards. (Of course, loads which may potentially access the same location as the pending store are not allowed to bypass in order to preserve correct program operation.)

The *floating-point unit* is fully pipelined with single-cycle throughput and three-clock-cycle latency for all single-precision instructions (except divide) and double-precision adds, subtracts, and compares. The first stage of the pipeline performs operand alignment and multiplication, the second stage performs addition, and the third stage rounds and normalizes the result. For double-precision multiplies and multiply-adds, the first pipeline stage is iterated twice, providing two-clock-cycle throughput and four-clock-cycle latency. Divides are nonpipelined and stall the floating-point unit for 18 clock cycles for single-precision operations and 33 clock cycles for double-precision operations.

The floating-point unit supports denormalized numbers in hardware. Since denormalized results require more time to process, a special non-IEEE mode provides a means to coerce denormalized results to zero. This eliminates data-dependent instruction execution time that would disrupt real-time data processing.

The primary function of the *system register unit* is to process the condition-register-logical instructions and the move operations to or from the SPRs. These instructions execute in one to three cycles, and are serializing in nature (that is, all preceding instructions must have been completed and retired). As a result they have somewhat less performance than might otherwise be expected. However, given the relatively infrequent use of instructions executed by the system register unit, the trade-off was made in favor of reduced complexity and silicon costs.

Completion buffer. The completion

Figure 4. PowerPC 603 microprocessor instruction flow

age of the
weakly or-
allows load
ing stores
due to load
ads which
same loca-
are not al-
to preserve

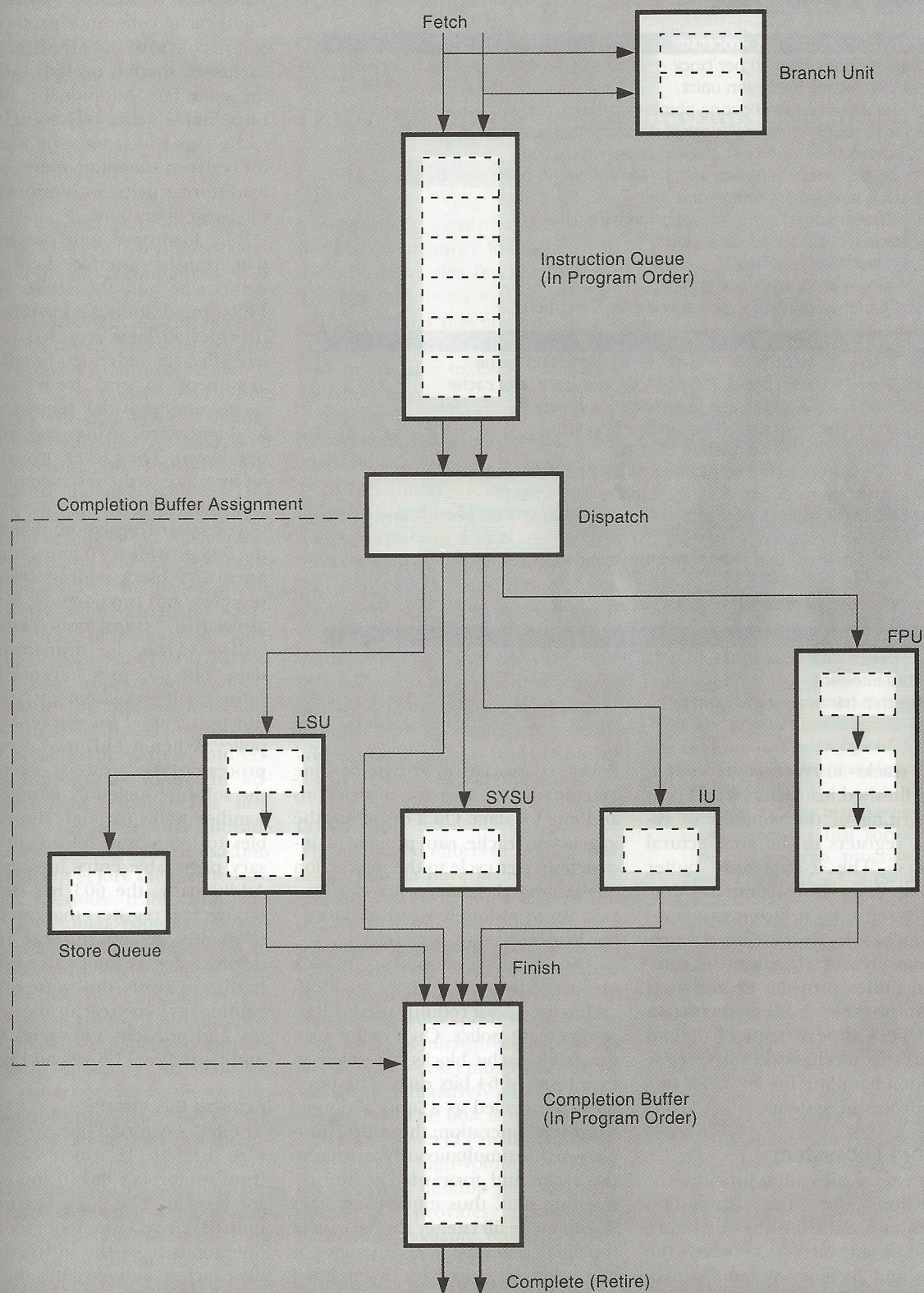
fully pipe-
throughput
ncy for all
ns (except
sion adds,
The first
orms oper-
tiplication,
s addition,
s and nor-
able-prec-
iply-adds,
is iterated
clock-cycle
k-cycle la-
elined and
nit for 18
ision oper-
for double-

it supports
hardware.
its require
pecial non-
eans to co-
ts to zero.
endent in-
that would
rocessing.

of the system
the condi-
ions and
or from the
execute in
are serializ-
preceding
been com-
result they
performance
e expected.
tively infre-
executed by
he trade-off
duced com-

completion

micropro-



603 Features

Superscalar Instruction Processor

- 2 instructions fetched per clock
- 5 independent execution units
 - branch, integer, floating-point, load/store, and system
- 3 instructions issued per clock (2 dispatched plus 1 branch)
- 2 instructions retired per clock plus branch folding
- 32, 32-bit general registers and 32, 64-bit floating-point registers
- 3-stage pipelined floating-point unit
 - floating-point denorm support in hardware
- 2-stage pipelined load-store unit
 - loads bypass stores
 - single entry store buffer
- Static branch prediction and speculative execution

Memory System:

- 8K, 2-way set associative, 32-bytes/line, instruction cache
- 8K, 2-way set associative, 32-bytes/line, copyback data cache
- Support for big-endian and little-endian addressing
- 64-entry, 2-way set associative, instruction TLB
- 64-entry, 2-way set associative, data TLB
 - hardware-assisted software table walk for TLB reloads
- 4 instruction block address translation registers (IBAT)
- 4 data block address translation registers (DBAT)
- 16 segment registers
- 64-bit, pipelined, split transaction, burstable external bus with parity
- 32-bit data bus option
- Internal clock multiplier 1x, 2x, 3x, 4x

System Features:

- Power Down modes (DOZE, NAP, SLEEP)
- JTAG Interface
- On-chip hardware debug support

buffer tracks instruction execution, retires finished instructions, and controls writing of the contents of renamed registers to the architectural registers. The completion buffer tracks up to five instructions at a time and can retire up to two instructions each clock cycle. Because instructions may execute out of order, the completion buffer provides an ordering mechanism that makes instruction completion appear sequential, and provides a mechanism for precise exception handling for PowerPC 603 microprocessor systems.

Memory Subsystem

The memory subsystem provides the instructions for the instruction fetcher and data for the load/store unit. Efficient access between the caches and memory systems is provided by the MMU, and the external bus interface.

Caches and Memory Management

The 603 incorporates two 8KB,

2-way set associative, 32-byte per line on-chip caches, one for instructions and one for data. On a cache hit, the instruction cache can provide 2 instructions per cycle to the instruction queue, and the data cache can provide up to a double-word of data to the load/store unit per cycle.

The data cache supports copy-back or write-through policies. Both caches use a least recently used (LRU) replacement policy. On a cache miss, the 603's cache blocks are filled in four beats of 64 bits each. The burst fill is performed as a 'critical-double word-first' operation; the critical double word is simultaneously written to the cache and forwarded to the requesting unit, thus minimizing stalls due to cache fill latency. In the case of the data cache, the burst operation is also used to write-back a modified cache line to memory.

Since the 603 was not specifically targeted for multiprocessing applications, the 603 design restricted certain aspects of data cache coherency

in order to save silicon area. Although the data cache only supports a three-state MEI (modified, exclusive, invalid) cache coherency protocol, it is compatible with full MESI (modified, exclusive, shared, invalid) caches on the same bus. Additionally, the data cache implements only a single set of cache tags which must be arbitrated for between snooping operations and load/store activity, with priority given to snoop operations.

The load/store unit provides the data transfer interface between the data cache and the GPRs and the FPRs. In addition the load/store unit provides all logic required to calculate effective addresses, handles data alignment to and from the data cache, and provides sequencing for load-and-store string and multiple operations. The caches provide a 64-bit interface to the instruction fetcher and load/store unit.

For faster translation of addresses, the 603 provides two four-entry, fully associative block address translation registers, and two 64-entry, 2-way set associative translation look-aside buffers (TLBs) for instructions and data. The 603 uses software to perform TLB replacements. A hash function is used for the replacement policy. When a TLB miss occurs, the processor takes a special exception to the software tablewalk handler. This handler walks through the page tables to locate and reload the necessary page table entry into the TLB. Additionally, the 603 has dedicated scratch pad registers that can be used to shadow general-purpose registers during software tablewalks. Through hardware assist, the entire tablewalk routine fits in two cache lines of memory and provides a low-cost, flexible, and fairly fast TLB reload capability.

External Bus Interface

The 603's external bus is compatible with the 601 [1], which was derived from Motorola's 88110 multiprocessor bus [8]. The 603's bus interface unit (BIU) receives requests for bus operations from the instruction and data caches, and executes the operations per the 603 bus protocol. Memory accesses can occur in single-beat (1 to 8 bytes) and four-beat burst (32 bytes) data transfers when the bus is configured as 64 bits, and in single-

With the flexibility built into the PowerPC Architecture, we expect to deliver a wide range of microprocessors all the way from low-cost embedded controllers through massively parallel supercomputers.

beat (1 to 4 bytes), two-beat (8 bytes), and eight-beat (32 bytes) data transfers when the bus is configured as 32 bits.

The BIU provides address queues, and prioritization and bus control logic. It consists of an independently arbitrated 32-bit address and 64-bit data buses (the data bus can optionally be configured as 32 bits). The 603's bus transaction consists of separate address and data tenures. This allows a variety of bus arbitration schemes to be supported. Specifically, the 603 supports address pipelining, where the address tenure of a new transaction is allowed to begin before the data tenure of the current transaction has completed. The 603's bus interface also supports split transactions, where the address and data tenures can be arbitrated for and controlled by different masters; and enveloped transactions, where the address and data tenures of a new transaction can occur after the address tenure of a previous transaction has ended, but before the data tenure for the previous transaction has begun. Enveloped transactions can be used for deadlock prevention in hierarchical bus environments, and to speed snoop push operations.

The 603 bus protocol also supports an address retry capability to support an efficient snooping protocol for memory coherency across the system. In a multiple processor system, the address retry can be used by a snooping master to interrupt another master's transaction on the bus. This becomes necessary when a bus master begins a transaction to access data that has been locally modified in the 603's cache. The 603 (referred to as the snooping 603) detects the access to this modified memory region, and uses the retry capability to force the first master to abort its transaction and retry it later. This enables the

snooping 603 to write back the modified data to memory for use later by the bus master that has been retried. The 603 bus interface also contains a clock multiplier that enables the processor to run at twice, three times, or four times the external bus clock speed.

Debug Features

The 603 incorporates a JTAG/IEEE 1149.1 boundary scan interface to facilitate board-level testing. The 603 also incorporates a special interface (accessible through the JTAG port) that allows an external service processor to read or write memory or any of the 603's internal registers. A special mode allows pipeline status information to be displayed for tracking the instruction stream in real time. Additionally, a programmable instruction address breakpoint is provided to assist in software debugging.

Conclusion

The combined efforts of Apple, IBM, and Motorola have been focused on creating PowerPC, a new RISC architecture that will form the basis of a whole new generation of high-performance, low-cost computers. With the flexibility built into the PowerPC Architecture, we expect to deliver a wide range of microprocessors all the way from low-cost embedded controllers through massively parallel supercomputers. The 603 is the first in a series of PowerPC microprocessors targeted at high-volume, low-cost, portable and desktop personal computers. It provides performance heretofore available only in significantly higher-priced, higher-cost, and higher-powered processors. ■

References

1. Allen, M. and Becker, M. Multiprocessing aspects of the PowerPC 601 Microprocessor. In *Proceedings of COMPCON* (Feb. 1993).

2. Becker, M.C., Allen, M.S., Moore, C.R., Muhich, J.S., and Tuttle, D.P. The PowerPC 601 Microprocessor. *IEEE Micro*, 13, 5 (Oct. 1993), 54-68.
3. Chang, A., Mergen, M.F., Rader, R.K., Roberts, J.A., and Porter, S.L. Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors. *IBM J. Res. Develop.* 34 (Jan. 1990).
4. Cohen, D. On holy wars and a plea for peace. *IEEE Comput.*, 14, 10 (Oct. 1981).
5. Dubois, M.C. and Scheurich, Briggs, F. Synchronization, coherence, and event ordering in microprocessors. *IEEE Comput.*, 21, 2 (Feb. 1988), 9-21.
6. ANSI/IEEE-1985 Standard for binary floating-point arithmetic, IEEE, Piscataway, N.J., 1985.
7. Groves, R.D. and Oehler, R.R. RISC System/6000 processor architecture. *Microprocessor & Microsystems* 14, 6 (July/Aug. 1990).
8. Gullette, B. The Design of the 88110 Bus Interface. In *Proceedings of RISC '92* (Feb. 1992).
9. Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28, 9 (Sept. 1979), 241-248.
10. Oehler, R.R., and Groves, R.D. IBM RISC System/6000 processor architecture. *IBM J. Res. Develop.* 34, 1 (Jan. 1990), 23-36.
11. Patterson, D.A., and Ditzel. The case for the reduced instruction set computer. *Computer Architecture News* (Oct. 15, 1980).
12. Patterson, D.A. and Sequin C.H. RISC-1: A Reduced Instruction Set VLSI Computer. In *Proceedings of the Eighth Annual International Symposium on Computer Architecture* (May 1981), pp. 443-457.
13. *The PowerPC Architecture*. Morgan-Kaufmann, San Mateo, Calif., Dec. 1993.
14. Radin, G. The 801 Minicomputer. In *Proceedings of the Symposium on Architectural Support for Programming Languages* (March 1982), pp. 39-47.
15. Simpson, R.O. and Hester, P.D. The IBM RT PC ROMP processor and

memory management architecture. *IBM Syst. J.* 26 (1987), 346-360.

16. Stone, J.M. and Fitzgerald, R.P. An overview of storage in PowerPC. IBM Res. Rep. RC 19133, Aug. 1993.

About the Authors:

BRAD BURGESS is a member of the technical staff at Motorola involved in both instruction set architecture and microarchitecture implementation. Current research interests include microarchitecture implementation and optimization, performance analysis, and compiler optimization. email: bradb@ibm.com

NASR ULLAH is a RISC applications designer in Motorola's RISC microprocessor division who has been working on PowerPC microprocessor board designs. Current research interests include parallel processing, computer design, computer architecture, performance measurement, evaluation and prediction. email: nasrullah@riscgate.sps.mot.com

PETER VAN OVEREN is a RISC applications designer at Motorola who has been developing user manuals and associated documentation for the PowerPC microprocessor family. Current research interests include electronic information distribution, computer architecture, performance evaluation, and systems design. email: pvo@daffysps.mot.com

Authors' Present Address: Motorola, 6501 William Cannon Drive West, Austin, TX 78735-8598.

DEENE OGDEN is a senior engineer at IBM's Somerset Design Center who was the 603 control logic team leader. Current research interests include microprocessor and computer design. **Author's Present Address:** International Business Machines Corporation, 11400 Burnet Road, Austin, TX 78758; email: deene@austin.ibm.com

In this article, the terms "PowerPC 603 Microprocessor" and "603" are used to denote the second microprocessor from the PowerPC Architecture family. PowerPC, PowerPC Architecture, PowerPC 601, PowerPC 603, and POWER are trademarks of the International Business Machines Corporation, and are used under license from International Business Machines Corporation. IBM is a registered trademark of International Business Machines Corporation. Motorola is a trademark of Motorola, Incorporated. Apple is a trademark of Apple Computer Corporation. SPEC, SPECint92 and SPECfp92 are trademarks of Standard Performance Evaluation Corporation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/94/0600 \$3.50

DO YOU HAVE YOUR ELECTRONIC MAIL ADDRESS ON ACM.ORG YET?



Setting up a Mail Forwarding (only \$10) or Full Service account on ACM.org is as easy as Emailing ACM Network Services at: Account-Info@ACM.org or calling 1-817-776-6876.



Or if you prefer, write:
ACM Network Services
P.O.Box 21599,
Waco, TX 76702 or
Fax: 1-817-751-7785.

THOUSANDS OF YOUR COLLEAGUES TURN TO ACM's TRANSACTIONS JOURNALS FOR THE CRITICAL INFORMATION THEY NEED ON:

- ◆ Computer Systems
- ◆ Database Systems
- ◆ Graphics
- ◆ Information Systems
- ◆ Programming Languages and Systems
- ◆ Mathematical Software
- ◆ Modeling and Computer Simulation
- ◆ Networking
- ◆ Software Engineering and Computer Simulations



To subscribe to any or all of ACM's Transactions Journals, call ACM's Member Services Department:

1-212-626-0500 (in metro NY and outside U.S.)
1-800-342-6626 (in U.S. and Canada)
Fax: 212-944-1318
Email: ACMHELP@ACM.org or write: ACM
1515 Broadway, New York, NY 10036.

The Making of the PowerPC



PowerPC 603 Microprocessor Power Management

Addressing the need for long battery life in portable applications and environmental concerns about energy consumption requires microprocessors with low power consumption as well as high performance. A primary design goal of the PowerPC 603 microprocessor was to provide sophisticated power management without compromising next-generation performance. As a result, the 603 is ideal for portable applications such as laptop computers in addition to Energy Star compliant desktop computers. The PowerPC 603 allows the system designer to control energy consumption through both hardware and software means as well as providing automatic internal power management.

PowerPC 603 Features

The 603 contains numerous power management (PM) features:

- 0.5 μ m four-layer metal CMOS process
- 3.3V power supply
- 5V or 3.3V compatible inputs
- Functional units can be dynamically powered up and down
- Four power states
- Fully static design
- Reduced-pinout mode
- 64-bit or 32-bit data bus

Dynamic Power Management

A sophisticated feature of the PowerPC 603 is dynamic power management (DPM), which automatically powers up and down the individual execution units on the chip, based on the contents of the instruction

stream. For example, if no floating-point instructions are being executed, the floating-point unit is automatically powered down. Power is not actually removed from the execution unit; instead, each execution unit has an independent clock input, which is automatically controlled on a clock-by-clock basis. Since CMOS circuits consume negligible power when they are not switching, stopping the clock to an execution unit effectively eliminates its power consumption. Setting a bit in an internal hardware register enables DPM.

Static Power States

The most unique feature of the 603 is the inclusion of four power states: Full On, Doze, Nap, and Sleep. Software selects these modes by setting one (and only one) of the three

power-saving mode bits. The 603 provides a separate interrupt and interrupt vector for power management: the System Management Interrupt (SMI). The 603 also contains a decrement timer that allows it to enter the Nap or Doze mode for a predetermined amount of time and then return to Full On operation through the Decrementer interrupt (DI). Note that the 603 cannot switch from one power management mode to another without first returning to Full On mode. The Nap and Sleep modes disable bus snooping; therefore a hardware handshake is provided to ensure data cache coherency before the 603 enters these power management (PM) modes. Table 1 summarizes the four power states, which are further detailed in the following paragraphs.

Table 1. PowerPC 603 power management states

PM mode	Functioning units	Activation method	Full On Wake Up method
Full On (with DPM)	Requested logic by demand		
Doze	Bus snooping Data cache as needed Decrementer timer	Controlled by software	External asynchronous interrupts Decrementer interrupts Reset
Nap	Decrementer timer	Controlled by hardware and software	External asynchronous interrupts Decrementer interrupts Reset
Sleep	None	Controlled by hardware and software	External synchronous interrupts Reset

Full On without DPM. Full On mode without DPM provides:

- Default state
- All functional units are operating at full processor speed at all times

Full On with DPM enabled. Full On mode with DPM provides on-chip power management without affecting the functionality or performance of the PowerPC 603.

- Required functional units are operating at full processor speed
- Functional units are clocked only when needed
- No software or hardware intervention required after mode is set
- Software/hardware and performance transparent

Doze. Doze mode disables most functional units but maintains data cache coherency by enabling the bus interface unit and snooping. A snoop hit will cause the PowerPC 603 to enable the data cache, copy the data back to memory, and then disable the cache.

- Most functional units disabled
- But snooping and time base/decrementer still enabled
- Several methods of returning to Full On mode
 - assert INT, SMI, or DI interrupts
 - hard or soft reset
- Transition to Full On state takes no more than a few processor cycles
- PLL running and locked to SYSCLK

Nap. The Nap mode disables the PowerPC 603 but still maintains the Phase Locked Loop (PLL) and the decrement timer. The timer can be used to wake up after a programmed amount of time. Because bus snooping is disabled for Nap and Sleep mode, a hardware handshake using the Quiesce request and acknowledge signals are required to maintain data cache coherency (see Sleep mode).

- Time base/decrementer still enabled
- Most functional units disabled (including bus snooping)
- All nonessential input receivers disabled
- Several methods of returning to Full On mode
 - assert INT, SMI, or DI interrupts
 - hard or soft reset
- Transition to Full On takes no more than a few processor cycles
- PLL running and locked to SYSCLK

Sleep. Sleep mode consumes the least amount of power of the four modes, since all functional units are disabled. To conserve the maximum amount of power, the PLL may be disabled and the SYSCLK may be removed. Before the PowerPC 603 enters the Nap or Sleep mode, the 603 will assert the QREQ signal to indicate that it is ready to disable bus snooping. When the system has ensured that snooping is no longer necessary, it will assert QACK and the

PowerPC 603 will enter the Sleep or Nap mode.

- All functional units disabled (including bus snooping and time base)
- All nonessential input receives disabled
 - Internal clock regenerators disabled
 - PLL still running (see following)
- Sleep mode sequence
 - set Sleep bit
 - 603 asserts quiesce request (QREQ_)
 - system asserts quiesce acknowledge (QACK_)
 - 603 enters sleep mode after several processor clocks
- Several methods of returning to Full On mode
 - assert INT or SMI interrupts
 - hard or soft reset
- PLL may instead be disabled and SYSCLK may be removed while in Sleep mode
- Return to Full On mode after PLL and SYSCLK disabled in Sleep mode
 - enable SYSCLK
 - reconfigure PLL into desired processor clock mode
 - system logic waits for PLL start-up and relock time (100 μ sec)
 - system logic asserts one of the sleep recovery signals (e.g., INT or SMI)

Power Management Mode Transitions

The PM mode transitions are typically controlled by the operating system (OS) or by the chipset hardware.

For example, as part of the OS loading/start-up process, one of the power-saving modes is selected by setting the appropriate power management mode bits. The proper power management mode is selected on the basis of whether any other devices will be performing DMA transfers that need to be snooped by the 603. If no snooping action is required, then Nap mode is chosen, otherwise Doze mode is used. If the time base is not needed either, or if its proper setting can be determined from an external source (such as a real-time clock) then the Sleep mode can be selected, offering the lowest power consumption. After setting the proper power management bit and creating whatever start-up processes are deemed necessary, the OS then creates a new low(est) priority process (an "idle" process). Now, when all the other processes are idle, this low priority process (or thread) is executed, placing the 603 in the selected power-down mode. Whenever the OS time counter (either the internal Time Base/Decrementer or an external countdown timer) expires and interrupts the processor, the 603 switches to Full On mode and processes the interrupt service routine. The interrupt service routine calls the OS scheduler to update priorities. If no other processes are active yet, the 603 then returns to the "idle" process and enters the low-power mode.

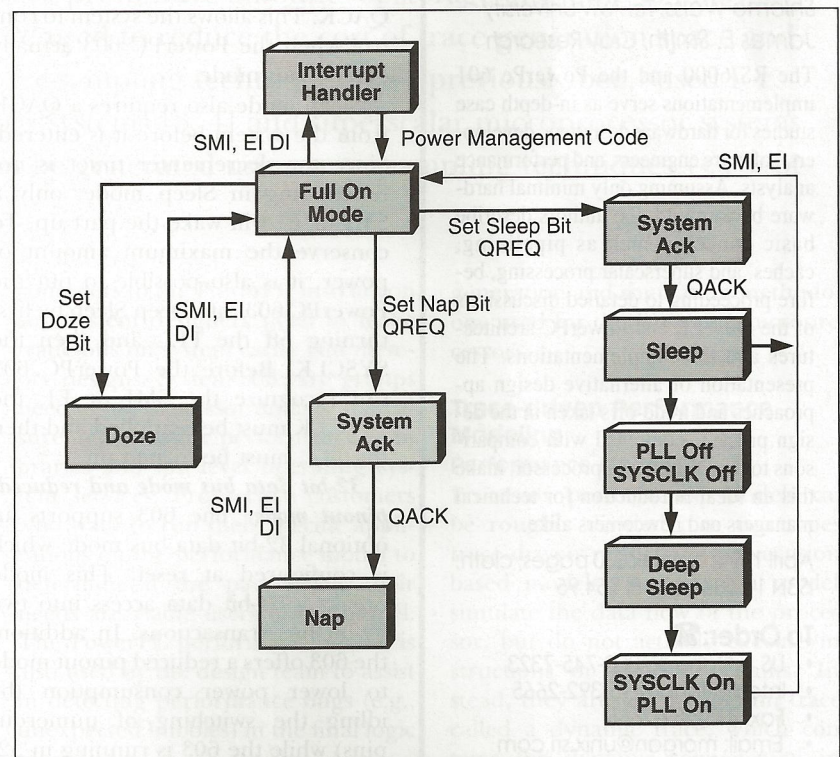
Another possible method is to rely on the PC chipset to monitor the bus activity of the 603 (similar to how most x86-based PCs currently operate). When activity timers in the chipset detect no activity by the system's components for a predetermined length of time, the chipset generates a SMI interrupt. In the interrupt handler for this routine, the 603 checks the interrupt status register in the chipset, indicating whether the interrupt was caused by a CPU or a peripheral activity timer expiring. If it was a CPU activity timer expiring, the 603 calls a routine that checks the status of the scheduler to decide whether or not the 603 was truly inactive (the 603 could have been simply executing a tight loop whose code and data spaces fit inside the on-chip caches). If the 603 was indeed inactive, the routine would then place the

Table 2. Typical power consumption

PM Mode	Frequency (Internal)		
	50 MHz	66 MHz	80 MHz
Full On no DPM	1.78 W	2.18 W	2.54 W
Full On with DPM	1.52 W	1.89 W	2.20 W
Doze	242 mW	307 mW	366 mW
Nap	89 mW	113 mW	135 mW
Sleep	66 mW	89 mW	105 mW
Sleep no PLL	15 mW	18 mW	19 mW
Sleep no PLL on SYSCLK	<5 mW	<5 mW	<5 mW

Table 3. 603 Performance

Benchmark	Frequency (Int/Ext) in MHz		
	50/50	60/66	80/40
SPECint92	47	60	75
SPECfp92	53	70	85

**Figure 1.** PowerPC 603 power management mode transitions

POWERPC

The IBM PowerPC Architecture: A Specification for the New Family of RISC Processors

IBM

The *IBM PowerPC Architecture* is the official detailed technical description of the IBM PowerPC architecture and its hardware conventions, making it an essential reference for designers of hardware and system software as well as application programmers developing products for any implementation of the PowerPC family of RISC microprocessors. It includes the base instruction set, storage model and all related facilities available to application programmers, the Time Base as seen by the application programmers and a full description of the system instructions.

June 1994; approx 600 pages;
cloth; ISBN 1-55860-316-6; \$49.95

IBM Power and PowerPC: Principles • Architecture • Implementation

Shlomo Weiss, Tel Aviv University
James E. Smith, Cray Research

The RS/6000 and the PowerPC 601 implementations serve as in-depth case studies for hardware designers, developers, software engineers, and performance analysts. Assuming only minimal hardware background, the authors describe basic concepts such as pipelining, caches, and superscalar processing, before proceeding to detailed discussions of the POWER and PowerPC architectures and their implementations. The presentation of alternative design approaches and trade-offs taken in the design process, combined with comparisons to the DEC Alpha processor, make this an ideal introduction for technical managers and newcomers alike.

April 1994; approx 600 pages; cloth;
ISBN 1-55860-279-8; \$54.95

To Order: ☎

- ♦ US & Canada 800-745-7323
- ♦ International: 415-392-2665
- ♦ Fax: 415-982-2665
- ♦ Email: morgan@unix.sri.com

Morgan Kaufmann
340 Pine St. 6th Floor
San Francisco, CA 94104-3205

603 in one of the PM modes, after checking the power status of the other peripherals in the system. It should be placed into the doze mode if any other peripherals with DMA capability are still powered up. If no other peripherals with DMA capability are powered up, or it is known that no DMA activity will take place, then the 603 can be placed into either Nap or Sleep mode.

The power management mode transitions of the PowerPC 603 are shown graphically in Figure 1. Depending on the power management scheme used, a decremented interrupt, SMI, or external interrupt (EI) invokes the interrupt handler routine. To enter the Doze mode, the doze bit is set. The PowerPC 603 will remain in Doze mode until an interrupt returns it to Full On mode. The PowerPC 603 then invokes the interrupt handler to determine its next action.

Transitioning to Nap mode requires a hardware handshake, since the PowerPC 603 will not enforce data cache coherency in this mode. Once the nap bit has been set and the PowerPC 603 is ready to Nap, it indicates this by asserting QREQ. When the system has assured that snooping is no longer required, it asserts QACK. This allows the system to control when the PowerPC 603 actually enters Nap mode.

Sleep mode also requires a QACK from the system before it is entered. Since the decremented timer is not functioning in Sleep mode, only a SMI or EI will wake the part up. To conserve the maximum amount of power, it is also possible to put the PowerPC 603 into Deep Sleep by first turning off the PLL and then the SYCLK. Before the PowerPC 603 can recognize the SMI or EI, the SYCLK must be reapplied and then the PLL must be turned on.

32-bit data bus mode and reduced-pinout mode. The 603 supports an optional 32-bit data bus mode which is configured at reset. This mode breaks a 64-bit data access into two 32-bit bus transactions. In addition, the 603 offers a reduced pinout mode to lower power consumption (by idling the switching of numerous pins) while the 603 is running in 32-bit data bus mode.

PowerPC 603 power consumption.

How all these power management features benefit the portable system designer is perhaps illustrated more clearly by the (preliminary) typical power consumption figures given in Table 2.

Power PC 603 performance estimates. Table 3 lists the simulated benchmark results for the 603, assuming a 1MB second-level cache.

Summary

The power management features of the PowerPC 603 enable portable systems designers to provide enhanced performance for portable computers while conserving energy. ☐

References

1. Motorola Inc. *Power PC™ 603 RISC Microprocessor. Tech. Sum.*—Rev. 1, 1993.

About the Authors:

BRAD SUESSMITH is an applications engineer with Motorola's RISC microprocessor division. Current research interests include high-performance microprocessor architecture, advanced personal computer architecture, and low-power/portable system design.

GEORGE PAAP III is an applications engineer with Motorola's RISC microprocessor division. Current research interests include microprocessor architecture, PowerPC assembly programming, multimedia authoring, rendering.

Authors' Present Address: Motorola, Inc., 6501 William Cannon Drive West, Austin TX 78735-8598; email: {brad_suessmith, george_paap}@email.sps.mot.com

In this article, the terms "PowerPC 603 Microprocessor" and "603" are used to denote the second microprocessor from the PowerPC Architecture family.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/94/0600 \$3.50

Ali Poursepanj

The Making of the PowerPC



The PowerPC Performance Modeling Methodology

The PowerPC performance modeling was based on trace-driven simulation, where the microprocessor organization is specified as a model, benchmark traces are generated and applied to the model, and performance data is measured and analyzed. In spite of the advantages of trace-driven simulation, which will be described later, meaningful benchmark traces are large and require prohibitive amounts of storage and time to analyze. Trace-sampling techniques were used to reduce the cost of trace generation space and simulation execution time. Trace-sampling techniques have previously been used for evaluation of the cache memory systems [3, 4] and superscalar microprocessor systems [5, 6]. Much smaller traces can be generated using trace-sampling techniques (<1% of the actual size) [4].

There are a number of advantages to a trace-driven approach. There is no need to generate program results in a trace-driven model. Designers can concentrate on performance implications of architectural features without having to worry about generating correct results or modeling system I/O and other overhead. Traces can be statistically sampled to reduce simulation time with a very acceptable reduction in accuracy. This allows relatively quick analysis of large benchmarks, such as the SPEC 92 suite.

The usefulness of the PowerPC performance models does not end at the processor design phase. Compiler developers need models to help tune

compilers to the PowerPC instruction set. System designers need to make trade-offs on system cache and memory designs. System software groups need to use processor models to measure and tune the performance of libraries and low-level operating system services. Prospective customers who want to run their specific applications on the performance model to determine if the part meets their needs are viable users of the model. The PowerPC performance model is also used by the design team to assist in detecting performance bugs (e.g., unexpected bubbles) in the final logic model.

The following sections will discuss performance modeling and the trace

generation and simulation methodology used for the PowerPC microprocessors.

Trace-driven Performance Modeling Performance Models

Processor performance models can be roughly divided into two types: trace-driven models and execution-based models. Trace-driven models simulate the data flow of the processor, but do not actually execute instructions or generate results. Instead, they are fed a program trace, called a dynamic trace, which contains the dynamic sequence of instruction addresses, instruction opcodes, and data addresses that

Trace sampling turned out to be a good technique for increasing the productivity of performance modeling.

occurred during execution. Execution-based models, on the other hand, execute code much as a real processor would, generating results and using those results in conditional branches and so forth. The benefits and drawbacks of each simulation model are described in the following paragraphs.

Although the trace-driven model offers advantages listed in the previous section, meaningful traces can require a large amount of storage and must be recorded from an existing system that has the same instruction set. Correctness of the traces or model behavior may be difficult to verify, since a trace-driven model does not generate program results. Data-dependent timing situations cannot be modeled in a trace-driven model.

In an execution-based model, program executables take up much less space than dynamic traces and do not require an existing system for trace generation. Correctness of the model behavior is easier to verify, since the model must generate the correct program results. Therefore, microarchitectural logic problems can be discovered and corrected at an earlier stage in the design cycle. Execution-based models provide more accuracy than trace-driven models and can execute faster than trace-driven models. A drawback to this method is that large benchmarks take a long time to run, since they require the full processor to be modeled, instead of just the data flow model. The longer run time could seriously deduct from productivity in making performance-oriented design trade-offs. Table 1 compares some of the features of the execution-based and trace-driven models.

Trace Generation

Instruction traces, the records of the sequence of memory addresses referenced by a processor during execution, are widely used by designers

and researchers to analyze the performance of computer systems. Trace generation refers to the process of recording the executed instructions. To generate an instruction trace, programs are compiled and executed, and traces are recorded during the program execution in a variety of ways. Traces can be generated through hardware-based monitors, interrupt-based methods, simulation-based methods, microcode-based methods, or instrumented program-based methods [9].

Hardware-captured traces. The processor's references to memory are recorded directly by a hardware performance monitor logic. The captured references are typically physical memory references, which are desirable for the cache memory simulation but are not as useful as virtual addresses for finite cache processor performance issues. Complexity, cost, and lack of flexibility are the primary limitations of this approach.

Interrupt-based traces. In interrupt-based techniques, the program is interrupted after execution of each instruction, and instruction address information is recorded. The problem with this approach is that the need for interrupting every instruction slows down the trace generation. Another problem is that tracing operating system code can be difficult.

Simulation-based traces. The simulation-based method is based on the use of a cycle-level architectural simulator that executes the simulated machine at the binary level. By implementing the 'programmer model' of the machine exactly, real workloads (both operating system and applications) can be traced. By implementing a cycle-level model, not only CPU memory accesses, but phenomena such as system bus, I/O bus, processor and I/O cache and buffer activities can be traced in detail. However, the speed of trace collection in this method is very slow.

Microcode-based traces. The mi-

crocode-based method enables capturing of long traces for multiprogrammed user and operating system activities. The advantage of this method is that it is fast and allows trace generation of existing and figure processors. The problem with this approach is that many RISC processors do not use microcode and CISC chips have their microcode in their read-only memory (ROM), which is not modifiable.

Instrumented program-based traces.

In the instrumented-based trace generation, the program source code (operating system or application) is directly modified for trace generation. This method has attracted considerable attention and can speed up the trace-generation process.

An IBM internal interrupt-based trace-generation tool was used for trace generation of the SPEC 92 and Transaction Processing Council (TPC) benchmarks. SPEC 92 and TPC source code were compiled and executed on RISC System/6000 servers in the Somerset Design Center, and traces were recorded. The TPC traces were full continuous traces, while the SPEC 92 traces were sampled traces. Sampling techniques were used to increase the simulation turnaround time during the performance analysis of the PowerPC microprocessors. Trace sampling turned out to be a good technique for increasing the productivity of the performance modeling.

Trace Sampling for Reducing the Cost of Simulation

Trace sampling has been used in the last two decades to reduce the cost of storage and execution time of the trace-driven simulation. Much smaller instruction traces can be generated to represent the full continuous trace with much lower costs [6]. The "representative" accuracy of the sampled trace in terms of how well it represents the actual workload is an obvious concern. Experimental re-

sults [6] show that much smaller sampled traces can work as well as the continuous full traces for performance estimation while resulting in a significant cost reduction.

Table 2 compares sampled traces and continuous traces for the SPEC 92 integer benchmarks. The six SPEC 92 integer benchmarks were simulated using full and sampled traces and the simulation results were compared. As shown in this table, the difference between the geometric mean instructions per cycles (IPC), used to calculate the SPEC ratios, of the sampled traces and mean IPC of the continuous traces is less than 2%

(1.000 vs. 0.9807) for a simulation time cost reduction of more than 800X (39762 vs. 48).

Notice that although the overall IPC geometric mean is within 2%, the error margin for individual traces can go up to 13% (for *sc*). The reason for this difference is explained in more detail in [6]. It was noticed that the *sc* has a high percentage of conditional branch and high cache-miss ratios compared to other integer applications. Apparently, one-million instruction uniform sampling did not work for *sc* in terms of IPC the way it worked for the other SPEC 92 integer applications. Figure 1 shows the sam-

pled and continuous trace IPC for the SPEC 92 *li*.

Ideally, sampled traces and continuous traces should generate the same amount of IPC when both are simulated on the same processor model under the same conditions. This may not be the only condition for qualifying a sampled trace to be representative of a full trace. Simulation results in [6] show that not only mean IPC but a broad set of other performance data (such as instruction frequency mix, nature of the branches in the trace, percent of instruction stalls in the pipeline) can closely and acceptably correlate between sampled and

Table 1. Comparison of execution-based and trace-driven models

Metric	Trace-driven model	Execution-based model
Accuracy	Moderate	High
Execution speed	Low	High
Usability for design verification and code optimization	Low	High
Fast performance estimation capability of the standard benchmarks using sampled traces	High	Low
Capability for study of the future designs	High	Low
Usability for cache and CPU design trade-offs	High	Low

Table 2. Comparison of sampled and continuous traces of SPEC 92 integer applications

SPEC 92 integer benchmark	Continuous trace IPC	Sampled trace IPC	Continuous trace length	Sampled trace length	Continuous trace simulation time (minutes)	Sampled trace simulation time (minutes)
<i>compress</i>	1.0000	0.9801	80,537,933	1,000,000	672	8
<i>eqntott</i>	1.0000	1.0373	1,067,642,760	1,000,000	8,898	8
<i>espresso</i>	1.0000	1.0645	1,141,805,665	1,000,000	9,516	8
<i>gcc</i>	1.0000	0.9539	114,605,018	1,000,000	960	8
<i>li</i>	1.0000	0.9944	2,000,000,00	1,000,000	16,668	8
<i>sc</i>	1.0000	0.8669	365,641,328	1,000,000	3,048	8
CINT92	1.0000	0.9807			39,762	48

Compress is a text compression and decompression utility that reduces the size of the named files using adaptive Lempel-Aiz coding. *Eqntott* translates a logical representation of a Boolean equation to a truth table. The primary computation is sort operation. *Espresso* is a tool for the generation and optimization of programmable logic arrays (PLAs). It characterizes work in the EDA market and logic simulation and routing algorithm areas.

Gcc is a popular compiler that compiles 76 input files. This benchmark represents one phase (compile) of software development application. *Li* is a Lisp interpreter that solves nine queens' problems. The backtracking algorithm is recursive and represents object-oriented programming type problems. *Sc* is a spreadsheet benchmark that calculates budgets, SPECmarks, and 15-year amortization schedules [8].

full-trace simulation data for a class of applications such as SPEC 92 integer benchmarks.

Table 3 compares mean IPCs, number of branches, source of delays, and instruction frequency mixes for the sampled and continuous traces for the SPEC 92 benchmark *li*. These numbers have been calculated for a continuous trace simulation using a

finite cache and a one-million-instruction-trace sampled trace using a statistical cache for the *li* application. The characteristics listed in the table affect the performance of the PowerPC microprocessors and are used to check the representativeness of the sampled traces. That is, not only the IPC of the sampled and the continuous traces should be very

close, but their simulation behavior should be very similar as well.

Simulation Methodology

A simulation tool called Basic RISC Architecture Timer (BRAT) was developed to implement the trace-driven methodology for design trade-offs and performance estimation for the PowerPC microprocessors. BRAT is a trace-driven, parameterized, simulation tool that has been used for performance modeling of the superscalar-based PowerPC microprocessors at the Somerset Design Center in Austin [5]. BRAT covers a large superscalar design space, allows quick turnaround for designers' what-if analysis and provides appropriate accuracy. The BRAT tool has been used for performance modeling of the PowerPC 601, PowerPC 603, PowerPC 604, and PowerPC 620 microprocessors.

BRAT provides several hundred parameters to model variations in the PowerPC superscalar microprocessor designs. These parameters are used to model:

- Cache organization
- Bus attributes
- Execution unit pipeline
- Reservation stations
- Branch prediction schemes
- Order of instruction issue
- Order of instruction execution
- Order of instruction completion
- Register renaming
- Data dependency
- Register file read and write ports
- Synchronization and serialization
- Exception
- Others

Superscalar architecture concepts have been described in detail in [2]. Figure 2 shows a simplified data flow in a superscalar organization. Instructions are *fetched* from the instruction cache (or memory), *decoded*, *dispatched*, *executed*, and *retired* at the CPU clock boundaries. Instructions flow through the CPU pipeline stages, wait in various buffers and queues, (depending on the hardware organization and program characteristics), spend some number of cycles in the execution units (similar to being served by servers in a queuing system), and retired through a com-

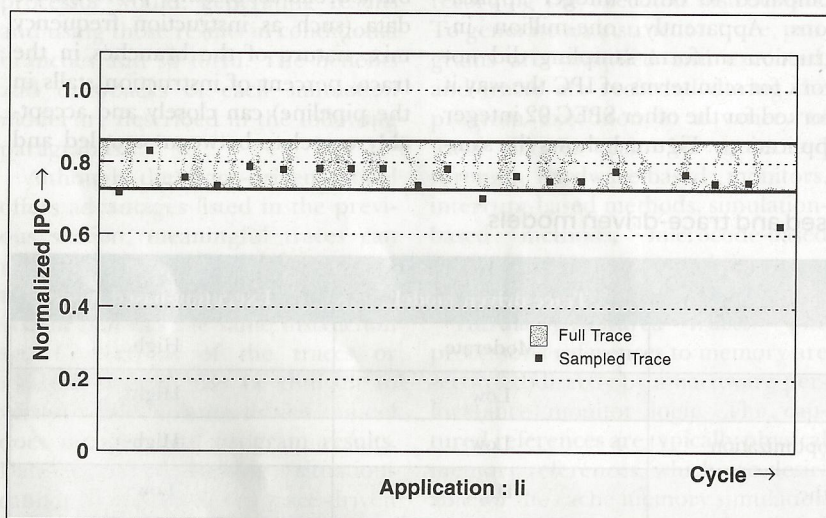


Figure 1. Sampled vs. Continuous Trace IPC for the SPEC 92 *li*.

This figure shows the sampled and continuous trace mean IPC vs. CPU cycle time for the SPEC 92 *li* application. The SPEC 92 *li* continuous trace contains nearly two billion instructions, where the *li* sampled trace contains only one million instructions. The continuous trace simulations used a finite cache model and the sampled trace simulation used statistical cache miss ratios taken from the full trace cache simulation. Notice that the continuous trace mean IPC reaches the steady state very quickly and the sampled trace mean IPC is very close to the continuous trace mean IPC. These characteristics have been seen on most of the SPEC 92 benchmarks [6] making the sampling of the SPEC 92 benchmarks very attractive for improving the productivity of the performance modeling.

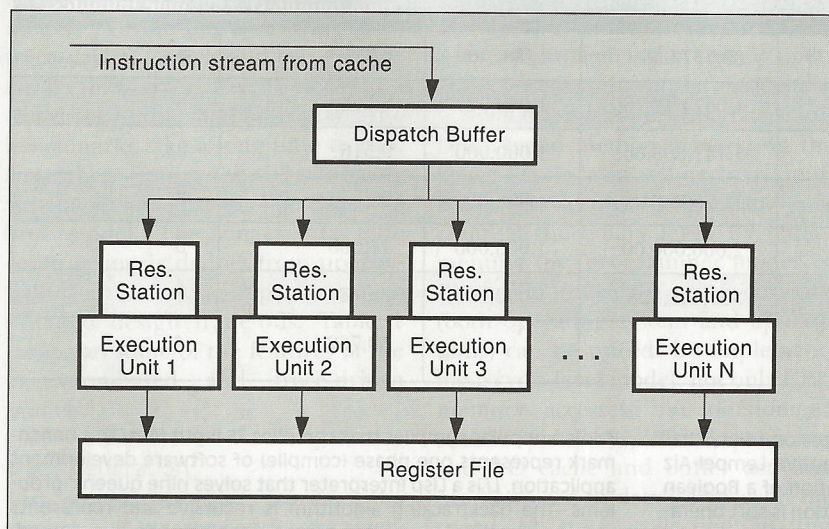


Figure 2. Instruction flow through a superscalar organization

pletion stage at some rate (leaving the system). In this environment the mean IPC of the processor can be viewed as the throughput of the queuing system.

BRAT allows modeling of a reconfigurable queuing system that correlates to various potential pipeline organizations in the superscalar design domain. In a simple queuing system, such as a drive-in bank, customers only wait if all the tellers are busy serving other customers. In the trace-driven simulation model, however, instructions may wait at different stages in the pipeline because of resource conflicts, incorrect speculative execution, data dependencies, serialization, cache misses, and many other reasons [2].

BRAT parameters can be classified in two major groups. The first group is parameters that are used to specify the hardware structure. These include: the number of execution units, the number of pipeline stages of the floating-point execution unit, the size

of cache, the size of the dispatch buffer, the instruction cache line size, and others. The second group is parameters that affect the flow of instructions and stalls through the pipeline. Some of these parameters are: order of execution, serialization rules, register-renaming, branch-prediction scheme, completion criteria, arbitration in a unified instruction and data cache.

BRAT accepts three inputs: 1) a dynamic instruction trace, representing the workload; 2) a parameter file, representing the CPU model; and 3) an architecture file, representing the supported instruction set. These inputs are specified either through the command line or through a graphical user interface. All the parameters defined in the parameter file can be overridden through the command line options. Having this feature in the BRAT allows scripts to be written that vary parameters so that performance trade-offs can be quickly explored.

The BRAT tool was made available to the PowerPC design teams from the beginning of the design stage for performance analysis and design trade-offs. A network of RISC System/6000 machines was used for running BRAT jobs in the Somerset Design Center. Figure 3 shows the general flow of a BRAT simulation.

As the detailed design of the PowerPC microprocessors evolved, it was very important to make sure the BRAT performance models continued to correlate with the designs. Special performance test cases were developed to verify the BRAT model against the logic model for the PowerPC processor [5]. There were three categories of performance test cases: 1) single instructions, in which each test case had one instruction; 2) instruction sequences with an instruction frequency distribution similar to those of the SPEC92 benchmark programs, and 3) selected inner loops of 10 of the SPEC92 benchmark programs. The selected inner-loops

Table 3. Comparison of the sampled and continuous performance data for *li*

Performance metrics	Continuous trace	Sampled trace	Difference
Mean IPC (completed instruction per CPU cycle)	1.0000	0.9944	-0.0056
Number of taken branches	1.0000	1.0183	+0.0183
Delays due to lack of resources	1.0000	1.0000	+0.0000
Delays due to incorrect branch prediction	1.0000	1.0714	+0.0714
Delays due to data dependency	1.0000	1.0000	+0.0000
Percentage of top 10 instruction frequencies in the trace			
Load(L)	22.33	22.42	0.09
Conditional Branch(BC)	14.34	14.36	+0.02
Store(ST)	9.95	9.69	-0.26
Compare Immediate(CMPI)	8.90	8.76	-0.14
Add Immediate(AI)	7.33	7.42	-0.09
OR Immediate(ORIL)	6.39	6.39	+0.00
Branch(B)	5.05	5.09	-0.04
Compare Logical Immediate(CMPLI)	3.25	3.22	-0.03
Compare Logical I(CMPL)	2.63	2.67	+0.04
Load Byte and Zero(LBZ)	2.59	2.49	-0.10

test cases proved most valuable, since they were actual benchmark code segments and uncovered design bugs related to performance.

The average IPC difference between BRAT simulations and logic model simulations when running the test cases listed previously was less than 5%. The difference was satisfactorily attributed to some cycle characteristics that could not be modeled exactly in the trace-driven simulation, such as predicted instruction paths not being taken, load/store misses of the wrong path, and data-dependent latencies.

Performance Estimation of Standard Benchmarks

SPEC 92, and Transaction Processing Council (TPC) applications are used as standard benchmarks by many computer manufacturers to measure the performance of their systems.

The SPEC 92 benchmarks consist of 6 integer and 14 floating-point applications [8]. TPC-A and TPC-B were the first two benchmark standards provided by Transaction Processing Council (TPC). TPC is the de facto industry standards group for on-line transaction processing (OLTP) performance benchmarks. TPC-A is a

superset of TPC-B. TPC-A can be described in terms of a hypothetical bank, which has one or more branches. Each branch has multiple tellers and many customers, each with an account.

SPEC 92 and TPC traces were used in the Somerset Design Center for performance estimation and design

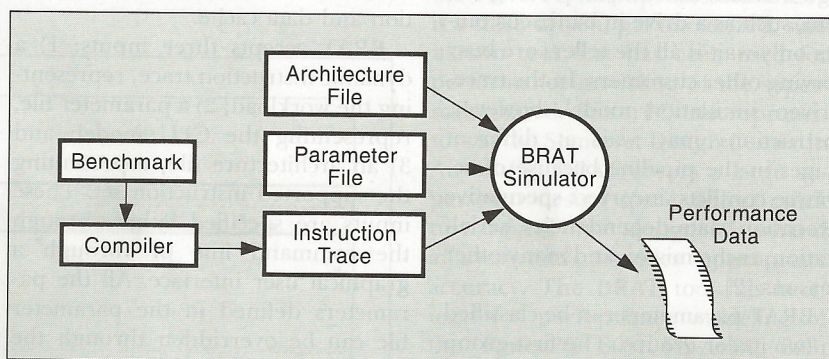


Figure 3. BRAT simulation environment

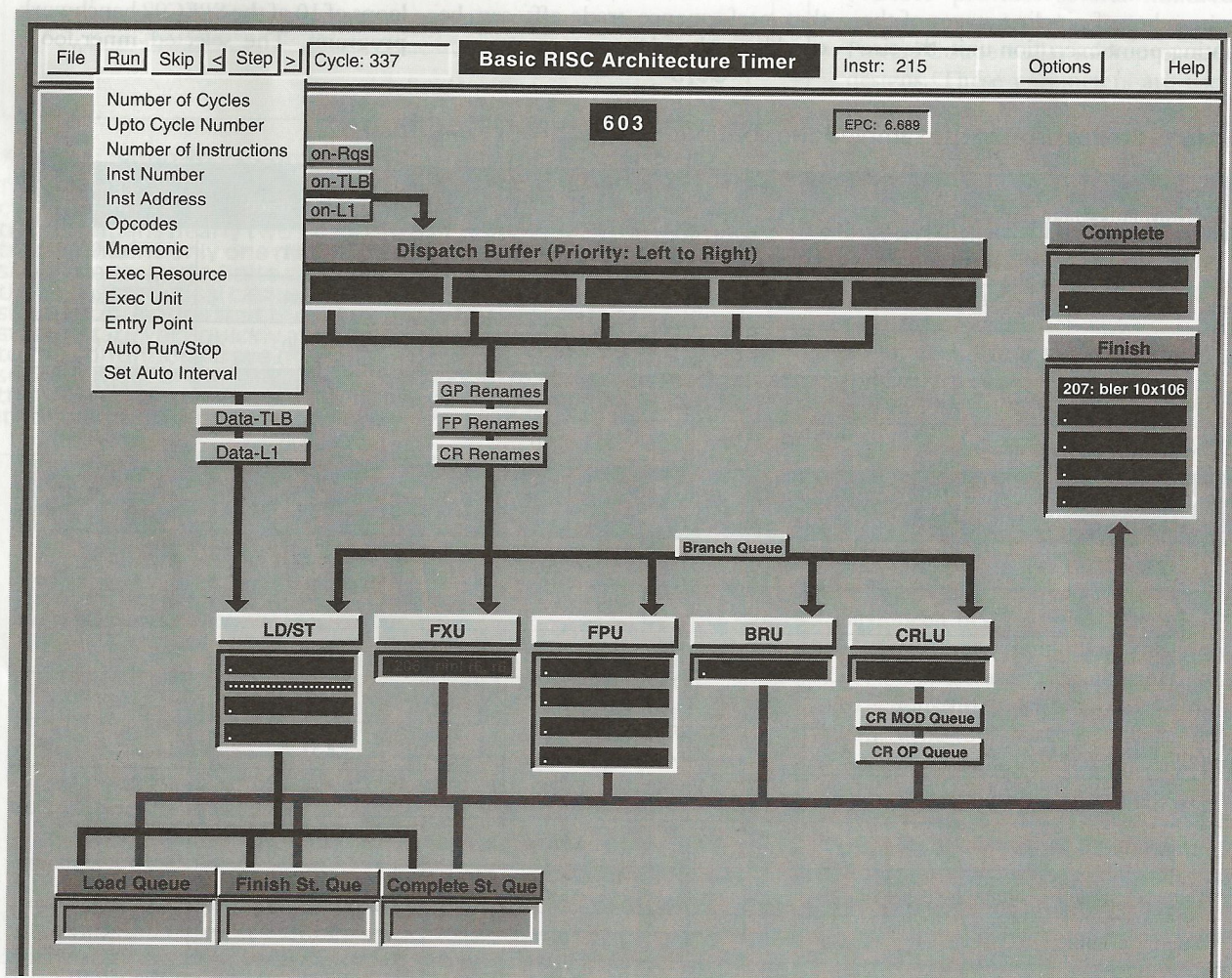


Figure 4. BRAT graphical user interface

Pipeline Report

Cycle	inst#	IB	X	FPU	LSU	B	C	Fi	Co
0	0	yz :							
1	1	zAB. : y							
2	3	BCD A : zy 							
3	4	CDEF B : z . . . A . Y .							
4	4	CDEFGH B : . z . . A . . .							
..	..	..	..	..	..	..	..	..	..

(IB = Instruction buffer, X = Fixed-point unit, FPU = Floating-point unit,
LSU = Load/store unit, B = Branch unit, C = CR unit, Fi = Finished, Co = Completed)

Inst.	Address	Opcode	Data Addr.	Mnemonic
Y	0x00000200	82420000	210c1c10	1 r18,0x0(r2)
Z	0x00000204	80f20008	210a5208	1 r7,0x8(r18)
A	0x00000208	39000000		li1 r8,0x0
B	0x0000020c	91070000	210c80a8	st r8,0x0(r7)
C	0x00000210	81320000	210a5200	1 r9,0x0(r18)
D	0x00000214	90690000	210c80a0	st r3,0x0(r9)
E	0x00000218	81320004	210a5204	1 r9,0x4(r18)
..	..	..	..	..

Cache Traffic Report

Cycle no.	R/S	instruction number	access type	address	penalty	instr 0 1 2	data 0 1 2	CB 1 2
i 0	+		fetch	00000200	0	h h	- -	-
i 1	+		fetch	00000208	0	h h	- -	-
i 2	+		fetch	00000210	0	h h	- -	-
d 3	+	0	load	210c1c10	0	- -	h h	-
i 3	+		fetch	00000218	0	h h	- -	-
i 4	+		fetch	00000220	0	h h	- -	-
d 4	+	1	load	210a5208	0	- -	h h	-
..	..	..	..	..	..	..	..	..

Symbol:

i=instruction, d=data

+ = request/served, r = request, s = serviced

h = hit, m = miss, b = busy, x = hold, \$ = forwarding data available, * = in service

Log Report

total cycles:	100
total inst completed:	64
Completed instructions per cycle (IPC)	0.64
* Dispatch Statistics	.
% cycles dispatch occurred:	51
% stall due to out of resources:	30
% stall due to mispred branches:	15
* Branch Statistics	.
total brs:	13
total guessed brs:	2
total guess right brs:	1
total guess wrong brs:	1
* Cache Access Statistics	.
* Completion Statistics	.
* Utilization of the Units	.
* System Parameters	.

Figure 5.
The pipeline,
cache traffic,
and log report
screen samples

trade-offs of the PowerPC microprocessors. One million instruction-sampled traces (200 samples of 5,000 continuous instructions), uniformly sampled over the entire application, turned out to be sufficient as well as efficient for performance estimation for each SPEC 92 benchmark [6]. Sampling of the TPC traces was not a trivial task, due to the nature of their cache memory accesses in which cache coldness effects at the start of the samples were significant. This resulted in unacceptable deviation of the sampled traces from the IPC of the full trace. Therefore, full traces were used for performance estimation and design trade-offs.

In order to further accelerate the process of the performance estimation, in addition to using the sampled traces, a distributed network of high-performance IBM RISC System/6000 servers was used for performing simulation jobs in the Somerset Design Center. Such an environment resulted in a new estimate of the processor performance for the designer's what-if questions using SPEC 92 benchmarks in less than 15 minutes. Without sampling and distributed computing, this time would be literally increased to days.

The BRAT tool was documented and made available to all the Somerset design team members from the beginning of the design cycle of the PowerPC microprocessors. More than 40 design engineers have used this tool to make at least one decision to assess performance implications when changes were required. BRAT will be enhanced and used for the performance analysis and design trade-offs of the future projects in the Somerset Design Center.

Other Applications

BRAT has also been used by other IBM PowerPC-based development organizations for cache and bus design trade-offs, and by the compiler optimization group to study the nature of the instruction sequences generated by the compiler. Cycle-by-cycle states of the CPU, as BRAT analyzes the instruction trace, can be visualized either on a graphical user interface or encoded in an ASCII file. This information can assist the compiler writer or system software devel-

oper in optimizing code for a target PowerPC implementation. Using the BRAT graphical interface, one can look at the state of the hardware elements at any cycle. Figure 4 shows a sample screen from the BRAT user interface. Users can set break points and conditionally stop the CPU execution when certain conditions occur. Some of these features are listed as follows:

- Watch the state of the machine cycle by cycle
- Run and stop at cycle number n
- Run and stop at instruction number n
- Run and stop at a specific address
- Run and stop at a specific instruction
- Run and stop at a specific entry point by name
- Backtrack n cycles
- Set a book mark at an address
- Return to an address by a book mark
- Others

In addition to the cycle-by-cycle state information displayed, BRAT produces several output files that contain detailed information about the run. The *log report* contains IPC, and accumulative nature of the branches, cache hit-and-miss ratios, load/stores, source of dispatch, execution, and completion delays, instruction frequency mix and utilization of all the resources in the system. The *CPU pipeline report* contains positional information of the instructions in the pipeline on a cycle-by-cycle basis. The *cache traffic report* contains cycle-by-cycle state information for cache requests. Figure 5 shows a sample screen output of the *log report*, *pipeline report*, and *cache traffic report*.

BRAT was also used in Somerset for the performance verification of the high-level logic model. The functionality of the high-level logic model was verified through billions of cycles of simulation. The cycle simulation in Somerset does an excellent job of covering functionality of the design, but it cannot easily detect the extra delays and bubbles in the design that affect performance. There can be situations in which the logic model is functionally correct but causes the instructions to wait extra cycles in the pipeline. BRAT was used to discover these

kinds of errors. In this process, the BRAT model and the logic model were compared on a cycle-by-cycle basis for performance-critical test cases. This was also used to verify the BRAT model as mentioned earlier.

An integrated version of BRAT is also available as the optional timer feature in the IBM PowerPC Visual Simulator (PVS). System designers can use this tool to study the memory organization of a proposed PowerPC-based system. PVS provides a PowerPC Architecture simulator to drive the timing simulator. Program executables are executed using the architecture simulator and the resulting program traces are fed to the timing simulator for performance analysis [7].

Summary

The PowerPC performance modeling was based on a trace-driven simulation approach. BRAT, a parameterized, trace-driven, and superscalar-based simulation tool was developed to define various organizations of the PowerPC chips. Sampled traces were used to reduce the cost and the turnaround time of the simulation. The representative of sampled traces were verified by comparing the outputs of the sampled and continuous trace simulation. The performance models were validated against the logic models throughout the design stage. Performance modeling tools were used by the design engineers for doing design trade-offs.

To enhance the quality of the current performance methodology, a combination of trace-driven and execution-based models is desirable for more accurate performance modeling. Design teams should work more closely with the compiler developers, operating system developers, and system designers from the beginning of the design cycle to achieve better system performance.

Acknowledgments

The author would like to thank Mike Peters for his assistance during the development of the BRAT tool, S. Surya for generating the SPEC 92 traces and doing cache simulation, Yanling Qi, for BRAT support, Robert Rosenblum and John Bunda for their assessment of an earlier version

of this article. Many thanks to Don Waldecker, Chin-Cheng Kau, David Levitan, Gary Veneski, Ray Dupont, Peter Song, Deene Ogden, Sonya Gary, Brad Burgess, Harry Dwyer, Seye Ewedemi, and David Lee, for their feedback and support during the project. 

References

1. Hennessey J., and Patterson, D. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, Calif., 1990.
2. Johnson, M. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, N.J., 1991.
3. Laha, S., Patel, J., and Iyer, R. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. Comp.* 37, 11 (1988), 1325-1336.

4. Liu, L., and Peir, J. Cache sampling by sets. *IEEE Trans. VLSI Syst.* 1, 2 (1993), 98-105.
5. Poursepanj, A., et al. The PowerPC 603 microprocessor: Performance analysis and design trade-offs. In *Proceedings of COMPCON 1994* (Feb. 1994).
6. Poursepanj, A., and Wu, C. Trace sampling for design trade-offs of microprocessors using SPEC 92 Integer benchmarks *IBM Tech. Rep.*
7. The PowerPC Visual Simulator for RISC System/6000, IBM Corporation.
8. SPEC 92 Release notes, System Performance Evaluation Corporation.
9. Stunkel, C., Janassens, B., and Fuchs, W.K. Address tracing for parallel machines. *IEEE Comput.* (Jan. 1991), 48-54.

About the Author:

ALI POURSEPANJ is an advisory engineer/scientist and team leader of the

PowerPC microprocessors group at IBM's Somerset Design Center. Current research interests include computer architecture, performance modeling, and trace sampling. **Author's Present Address:** IBM Somerset Design Center, 9737 Great Hills Trail, Austin, TX 78759; email: ali@ausvm6.vnet.ibm.com

IBM, PowerPC, PowerPC 603, PowerPC 604, PowerPC 620, PowerPC Architecture, and RISC System/6000 are trademarks of International Business Machines Corporation. SPEC is a trademark of System Performance Evaluation Corporation.

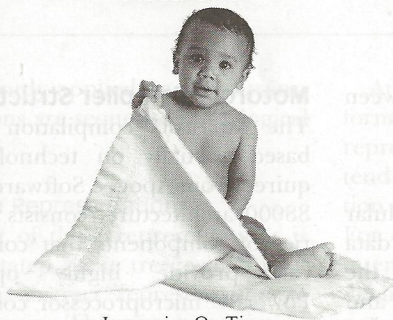
Permission to copy without fee all or parts of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/94/0600 \$3.50

We Can Think Of 9 Good Reasons To Immunize On Time.

Measles
Mumps
Diphtheria
Tetanus
Hepatitis B
Rubella
Spinal Meningitis
Pertussis
Polio

But You Only Need One.



Immunize On Time.
Your Baby's Counting On You.
Call 1-800-232-2522



U.S. Department of Health and Human Services

ACM LAUNCHES SIGMULTIMEDIA AND SOUND

SIGMULTIMEDIA provides a forum for researchers, engineers and practitioners in all aspects of multimedia computing, communication, storage and applications.

SOUND is a new electronic forum for the exchange of information on software, algorithms, hardware, and applications for digitally generated and/or manipulated audio signals.

For more information, contact ACM's Member Services Department, 1515 Broadway, New York, NY 10036

Phone: 1-212-626-0500
(in metro NY and outside U.S.)
1-800-342-6626
(in U.S. and Canada)
Fax: 212-944-1318
Email: ACMHELP@ACM.org.



The Making of the PowerPC



A Modular Approach to Motorola PowerPC Compilers

The need for balance between software and hardware is a well-known principle of RISC microprocessor design methodologies. In order to achieve a high level of performance, RISC microprocessors are designed to allow compilers to take full advantage of the pipelines and resources available. The PowerPC family of microprocessors is being designed to be used for many purposes, ranging from low-power embedded controllers to powerful, supercomputer-class multiprocessor systems. This diversity of uses will lead to an equally diverse set of operating system environments, including AIX, Macintosh OS, Solaris and Windows/NT, among others. Despite the multitude of PowerPC processor and system configurations being developed, there remains a need for highly optimizing compilers that utilize both the base PowerPC Architecture as well as other implementations of the chips and systems designed around the architecture.

Motorola has developed a highly optimizing, modular compilation environment that can be quickly adapted to various PowerPC microprocessor and system configurations. This suite of C, C++ and Fortran compilers is designed to meet the following criteria:

- Highly optimizing, ensuring optimal performance for PowerPC microprocessors
- Highly retargetable, ensuring rapid time-to-market
- Highly configurable, supporting multiple object file and debugging formats
- Compliant to software standards,

ensuring portability of code between chips

This article will describe the modular structure of the compiler, the data and information flow through the major phases of the compiler, and offer some discussion on architecture and implementation-specific optimizations currently performed in the PowerPC compilers. Through examples and descriptions, an understanding can be achieved as to how the Motorola PowerPC compilers are designed to provide the high performance and diversity that is essential to the PowerPC Architecture.

Motorola Compiler Structure

The Motorola compilation system, based partially on technology acquired from Apogee Software for the 88000 architecture, consists of a series of components that collectively can provide highly optimized PowerPC microprocessor code for a wide range of source languages, object file and debugging formats, and system environments. Conceptually, the heart of the Motorola compilation system is a common core that integrates multiple front ends with target-specific code generators to provide consistently high performance across an extremely wide range of target environments (see

Figure 1).

Additionally, the Motorola compiler is designed to maximize performance for RISC applications by providing three distinct internal representations of the program: a Tree representation, a RISC representation and a Machine representation. Collectively, these three representations create a foundation on which a series of program transformations and optimizations can be constructed. Each of the three representations is uniquely suited to a particular class of program transformations, ensuring that optimizations can

be efficiently applied and that transformations are sequenced in a logical manner.

Another common aspect of transformations performed on the Tree representation is that such changes tend to expose additional information for later phases of the compiler. For example, subroutine inlining or interprocedural analysis can expose details to the optimizer that would not otherwise be available. This analysis is accomplished by looking across subroutines, and, in some cases, file boundaries.

The Tree representation does not treat the program as a sequence of machine instructions, and, therefore, does not yet divide the program into basic blocks. With program control flow still explicitly represented as tree

operators, the compiler can quickly perform certain control flow optimizations to minimize branch instructions—an important consideration for superscalar RISC microprocessors such as PowerPC microprocessors. Explicit control flow information also simplifies the process of transforming program loops, and often helps identify loop induction variables which must be carefully analyzed as part of the loop transformation process.

The second representation is a set of instruction templates resembling a generic RISC instruction set. The pri-

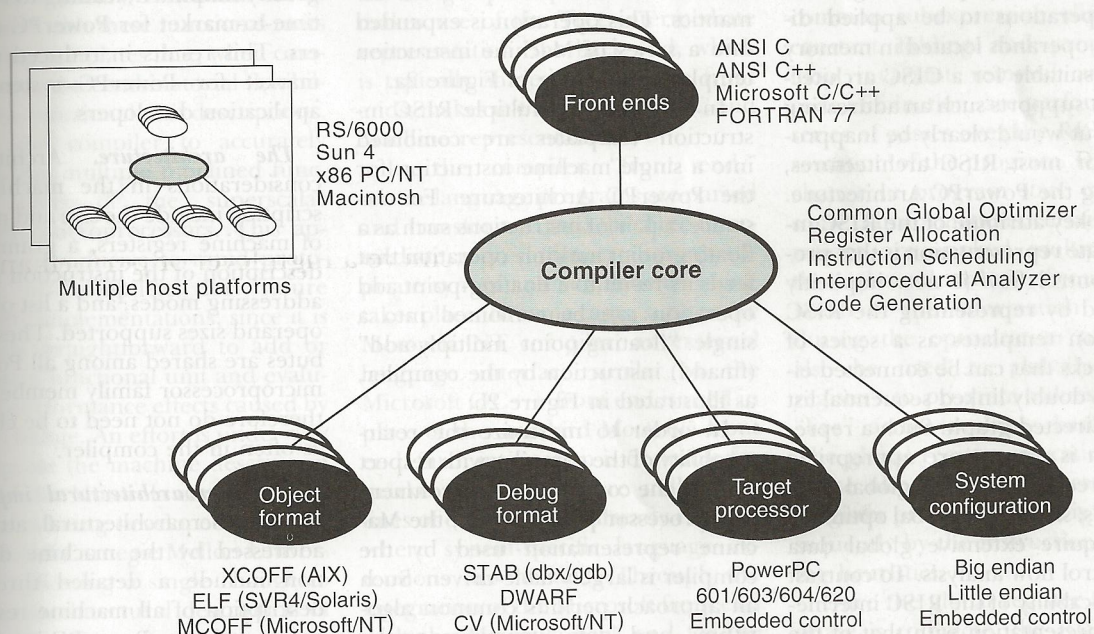


Figure 1. Overview of Motorola compilation system

mary purpose of this RISC representation is to permit the global optimizer to efficiently perform a series of aggressive performance-enhancing transformations without regard to the details of the underlying RISC microprocessor. Although it may seem unusual to ignore the details of the implementation when performing optimizations, doing so permits the optimizer to focus only on those aspects of a machine model that are relevant to RISC microprocessors. For example the RISC intermediate

representation assumes an unlimited number of registers, a load/store memory architecture, and arbitrary-size literal values. While such assumptions represent only estimates, they are particularly useful considering the design of RISC microprocessors. For example, the assumption of infinite register resources is appropriate when considering a modern compiler that is targeting the PowerPC Architecture, which has a total of 32 general-purpose registers and 32 floating-point registers.

Conversely, this assumption would be inappropriate for a CISC architecture that has only a few registers available for general use by the compiler. Additionally, permitting arithmetic operations to be applied directly to operands located in memory may be suitable for a CISC architecture that supports such an addressing mode, but would clearly be inappropriate for most RISC architectures, including the PowerPC Architecture. Another key attribute of the RISC intermediate representation is that program control flow is now implicitly described by representing the RISC instruction templates as a series of basic blocks that can be connected either as a doubly linked sequential list or as a directed graph. Such a representation is often more appropriate than a tree structure for global optimizations, since most global optimizations require extensive global data and control flow analysis. To contrast the applicability of the RISC intermediate representation with that of the previous Tree representation, consider that most transformations performed on the Tree representation are associated with program structure, whereas most RISC-level transformations are involved with instructions.

The third intermediate representation used by the compiler is an annotated version of the RISC representation. This representation contains additional information about machine resources, including registers and functional unit descriptions. Additionally, this representation is used by the compiler to perform target-specific transformations, including register allocation, instruction scheduling, and branch prediction.

The Machine representation is created by expanding the instruction templates of the RISC representation into instructions or sequences of instructions that have a 1:1 correspondence with the actual machine instructions of the target microprocessor. In some cases, a single RISC instruction template may expand to several PowerPC instructions. For example, the PowerPC Architecture does not include an instruction to perform integer absolute value. However, the RISC intermediate representation of the program treats "absolute value" as a single operation, in order to present the optimizer with a clear view of the intended program semantics. This operation is expanded into a series of Machine instruction templates, as shown in Figure 2a.

In other cases, multiple RISC instruction templates are combined into a single machine instruction for the PowerPC Architecture. For instance, a pair of instructions such as a floating-point multiply operation that feeds its result to a floating-point add operation can be combined into a single "floating-point multiply-add" (fmadd) instruction by the compiler, as illustrated in Figure 2b.

In order to maximize the retargetability of the compiler with respect to both the compilation environment and processor performance, the Machine representation used by the compiler is largely table-driven. Such an approach permits common algorithms and data structures to be shared for all environments while still permitting the code to be optimized for a particular target implementation. The target-specific data tables contain a great deal of information regarding the underlying microprocessor. The information typically addresses three target attributes: 1) the programming environment, 2) the architecture, and 3) the microarchitectural implementation.

The programming environment. The first type of information conveyed by the machine description tables involves the target programming and execution environment. The Motorola compilation system can support a number of object file formats, such as XCOFF and ELF, as well as several debugging formats,

including STAB (dbx), DWARF, and CodeView (NT). Each of these formats requires compiler support of some kind, and often requires different subroutine calling sequences or data layouts. For example, the Motorola NT compiler must target a little-endian environment that requires CodeView symbolic debugging information, whereas the Motorola C and Fortran compilers for AIX systems use big-endian, XCOFF conventions and a STAB-style debugging format. By describing such information in an abstract, table-driven manner the task of retargeting the compilers to a different combination of programming environment attributes is greatly simplified, leading to reduced time-to-market for PowerPC compilers. This results in reduced time to market for PowerPC systems and application developers.

The architecture. Architectural considerations in the machine description include the size and number of machine registers, a grammatical description of the instruction set and addressing modes, and a list of literal operand sizes supported. These attributes are shared among all PowerPC microprocessor family members, and therefore do not need to be changed as often in the compiler.

The microarchitectural implementation. Microarchitectural attributes addressed by the machine description include a detailed three-level description of all machine resources for a particular PowerPC implementation. This description is used extensively by the instruction scheduler to determine the optimal code sequence for a particular application on a specific PowerPC implementation.

The first level of microarchitectural description is a set of resource definitions. A resource can be virtually any aspect of the microarchitecture that is desirable to take into account when performing instruction scheduling. Examples include a functional unit pipeline stage, an instruction fetch or dispatch slot, or a read or write port to a register file. Resources can be singularly or multiply defined, allowing a fairly arbitrary set of RISC microprocessor attributes to be created as building blocks for a higher-level machine description.

A single RISC instruction template may expand to several PowerPC instructions.

For instance, the number of instruction issue slots differs between the 603 and 604 microprocessors. Consequently, this fact can be accurately described in the corresponding compiler machine description tables.

The second level of microarchitectural description is formed by combining the machine resources into a series of arbitrary functional units. Associated with these functional units are timing templates that, when combined with the combinatorial flexibility of the functional unit descriptions, enable the compiler to accurately portray the multiple pipelined functional units of the superscalar PowerPC microprocessors. This approach is conducive to rapid prototyping of compilers for future PowerPC implementations, since it is relatively straightforward to add or replicate a functional unit and evaluate the performance effects caused by such a change. An effort is under way to integrate the machine description used by the Motorola compiler with simulators being developed by Motorola [2] and Carnegie Mellon University [1], allowing a single change to the microarchitectural description of a PowerPC microprocessor to generate a new simulation model as well as new code that is tuned for that model. Such integration would allow for making more accurate and timely microprocessor design decisions.

The third level of the microarchitectural description used in the Motorola compilers involves the mapping of each machine instruction to one or more functional units. Such a mapping creates the link needed between the Machine representation of the code and the actual timing information required for accurate scheduling.

Phases of Compilation

The Motorola compilers are implemented as a series of interrelated phases that ultimately transform a

user's high-level source program into highly optimized PowerPC assembly code which can then be assembled and linked to form an executable program. Figure 3 outlines some of the major phases of the Motorola compilers. Traditionally, compilers have often been described in terms of a "front end" and a "back end." For the Motorola compiler, the front end makes use of the Tree intermediate representation; the back end, which is typically shared by multiple front ends, makes use of both the RISC and Machine representations.

In the compiler "front end," source language programs are parsed and transformed into the Tree intermediate representation after appropriate syntax and semantic checking takes place. In some cases, such as the Motorola NT compiler, an external language interface similar to the Microsoft C/C++ front end is used in place of the standard Motorola front end. Such integration is desirable for market segments where millions of lines of existing code already exist, or where system-specific language extensions are needed. Through the integration of the Microsoft front end with the Motorola back end for the NT compiler, existing Microsoft application code and the associated build structures (i.e., Makefiles) can be quickly ported to PowerPC systems with little or no source changes while still picking up the benefits of the highly optimizing Motorola back end.

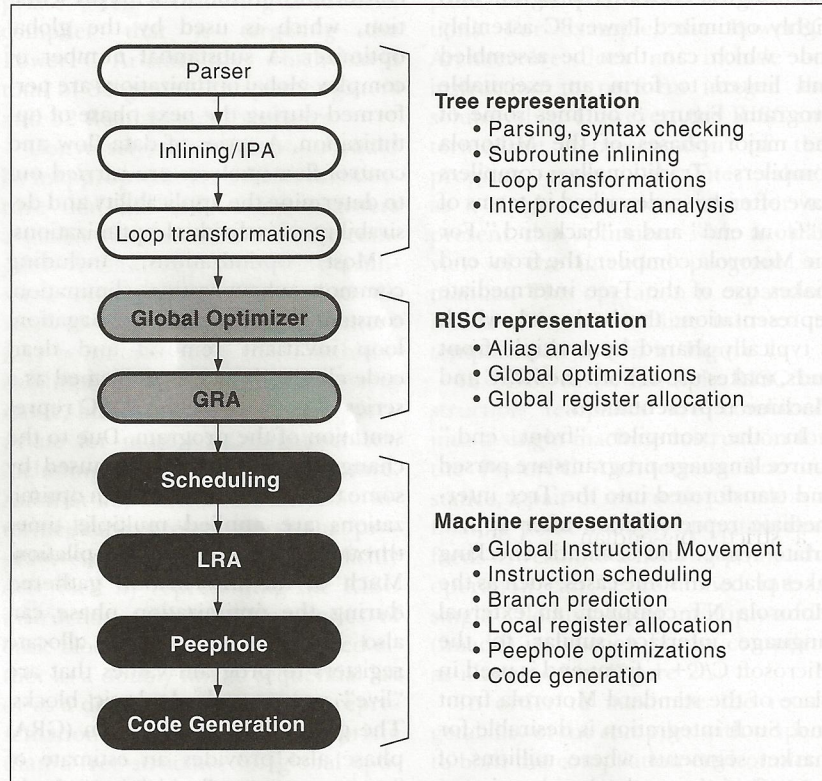
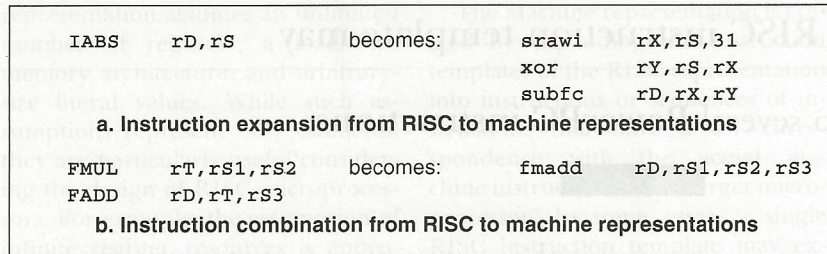
Optimizations performed during this phase of compilation include subroutine inlining, interprocedural analysis and a series of loop transformations, such as software pipelining and loop unrolling. These program transformations all share the common goal of improving performance by exposing more information to later phases of the compiler.

On completion of front-end processing, the Tree representation is

converted into the RISC representation, which is used by the global optimizer. A substantial number of complex global optimizations are performed during the next phase of optimization. A series of data flow and control flow analyses are carried out to determine the applicability and desirability of individual optimizations.

Most optimizations, including common subexpression elimination, constant folding and propagation, loop invariant removal and dead code elimination, are performed as a series of passes over the RISC representation of the program. Due to the change in program state caused by some transformations, certain optimizations are applied multiple times throughout this phase of compilation. Much of the information gathered during the optimization phase can also be used to globally allocate registers to program values that are "live" across multiple basic blocks. The global register allocation (GRA) phase also provides an estimate of "register pressure" which is used subsequently by the instruction scheduling heuristics.

Once the global register allocation phase is complete, the RISC representation is expanded into the Machine representation, which is used for the instruction scheduling and local register allocation (LRA) phases of compilation. Instruction scheduling is applied in two stages. First, instructions are globally moved across basic block boundaries to hide instruction latencies that may not be possible to hide within the original basic block. Such code motion is particularly valuable for "compare" instructions in the PowerPC Architecture, since there is often a significant delay between the generation of a condition code and its use in some PowerPC implementations. After global scheduling is complete, a local scheduling pass is implemented within each basic block. It is this



601 schedule		603 schedule	
lfs	f1, 0x0 (r4)	lfs	f1, 0x0 (r4)
lfs	f2, 0x0 (r3)	addi	r5, r5, 0x1
addi	r5, e5, 0x1	lfs	f2, 0x0 (r3)
fmuls	f1, f1, f2	cmpi	0x6, 0x0, r5, 0x1e
cmpi	0x6, 0x0, r5, 0x1e	fmuls	f1, f1, f2
stfsu	f1, 0x4 (r4)	stfsu	f1, 0x4 (r4)
bc	0xc, 0x18, L..2	bc	0xc, 0x18, L..2

Figure 2. Mapping RISC representation to machine representation

Figure 3. Compilation phases

Figure 4. Scheduling difference between a) 601 and b) 603 compilers

phase of compilation that exploits the superscalar nature of the PowerPC microprocessors.

Currently, the Motorola compilers can be directed by command line flags to schedule code for any of the 601, 603, 604 and 620 implementations. This process often leads to significant performance improvements, while still maintaining instruction set compatibility across all family members. Local register allocation follows

instruction scheduling for each basic block. Although there are trade-offs between the relative ordering of the scheduling and register allocation phases, the integration of scheduling between GRA and LRA helps minimize the potential difference in the resulting code. Finally, after a brief "peephole" optimization phase, assembly code is generated by the compiler and the compilation process is completed.

Compilation for PowerPC

The PowerPC Architecture is a full 64-bit RISC architecture that was derived from the IBM's existing POWER architecture [4]. In defining the PowerPC Architecture, the POWER architecture was simplified in several ways to remove aspects that might prohibit aggressive superscalar and high clock rate implementations. With respect to the compiler, there are several aspects of the PowerPC Architecture that distinguish it from other RISC architectures. The PowerPC Architecture contains several features that affect the compiler. These include the instruction set, special-purpose registers, the branch model and the memory model.

Instruction set. The PowerPC instruction set offers several key instructions that compilers can utilize to improve performance. One key type of instructions includes the path-length reduction operations such as "floating-point multiply-add" and "load/store with update." A second type of instructions includes a powerful set of bit-field operations that can sometimes be used to replace comparatively expensive branch instructions. Finally, special instructions are also included to perform logical operations on condition code results, once again leading to a decrease in branch instructions.

Special-purpose registers. The PowerPC Architecture has several special-purpose registers used by the compiler, including the Condition Code register, the Link register, and the Count register. The Condition Code register has eight fields that can hold the results of comparisons. These fields are usually set by compare instructions, or, optionally, by the record form of other instructions.

Conditional branch instructions most commonly use the condition code fields, although condition codes can also be moved to or from the general-purpose registers.

The Link register is used for subroutine linkage, as its name suggests. The Count register is used in several ways, though primarily to hold the index value in loops. The PowerPC Architecture provides a branch conditional form which decrements the value in the Count register, compares the result to zero, and then branches based on the result of the comparison. The compiler can make use of such a mechanism to eliminate both a decrement operation and a compare instruction.

A register sometimes reserved through software convention, the table of contents (TOC) register, is not part of the architecture, but it is defined by the PowerOpen Application Binary Interface [3] as part of the convention for linking. The TOC is a common storage area used to look up all address constants and external symbols for a given object.

Branch model. The conceptual model of the PowerPC Architecture defines the branch unit as an execution unit. It fetches instructions from the instruction cache and processes branches out of the instruction stream. It dispatches instructions to the appropriate execution units. Of the multiple forms of branch addressing employed by the PowerPC Architecture, only one has a direct consequence for the compiler. For performance reasons, it is preferable to use the Count register, rather than the Link register, for the generation of register-indirect branches, including, for instance, the implementation of a jump table or the invocation of a subroutine through a memory pointer.

Most implementations of the PowerPC Architecture implement some form of branch prediction in order to minimize potential delays associated with control flow instructions. The architecture defines an encoding of one bit in branch instructions to provide static (i.e., compiler driven) prediction hints to the microprocessor. If that bit is not set, then backward branches are predicted

taken, and forward branches are predicted not taken. Setting this bit in the encoding of the branch instruction will reverse the sense of the prediction. Some microprocessors implement dynamic prediction methods in which the static prediction bit may not be needed by the microprocessor.

Memory model. The PowerPC Architecture allows for a weakly ordered storage model. This gives microprocessors freedom to rearrange memory accesses relative to the order specified in the program stream. The processor must ensure data dependencies are not broken in order to preserve program semantics. The PowerPC Architecture offers user-mode cache control instructions for both uniprocessor and multiprocessor configurations. These instructions allow the programmer or compiler to increase performance by clearing, storing or preloading cache lines with data that may be needed soon in the instruction stream.

Whereas the POWER architecture is a strictly big-endian data access model, the PowerPC Architecture was extended to support both little- and big-endian modes. This mode can be dynamically switched and specifies that the little-endian transformation applies to both instructions and data. The architects provided full support for the PowerPC Architecture in multiprocessing environments. The memory model was enhanced in such areas as atomicity, coherence and aliasing in order to cover all aspects of multiprocessing environments.

PowerPC Microarchitectural Differences

Each implementation in the PowerPC Architecture has its own microarchitecture description. The microarchitecture defines the execution units, the pipelines, the bus, and the memory model of a microprocessor. The information about these aspects of a chip are not needed by the compiler to produce correct code, but can impact performance significantly. The Motorola compilers have run-time flags that will target each implementation in order to achieve the best performance for each member of the family.

Superscaler issue. One major area of

difference between implementations of the same architecture revolve around the instruction issue model. Each microarchitecture has different models for instruction issue as well as different execution units and instruction pipelines. This impacts how the compiler should schedule sequences of code. The scheduler uses the implementation-specific information from the machine descriptor tables to determine how each instruction will flow through the machine. It then tries to reorder instructions to avoid machine resource conflicts and to provide the result from one instruction by the time it is needed for any data-dependent instructions in the program. This is strictly a performance enhancement, and does not affect code compatibility. Figure 4 illustrates the difference in code schedules for two of the PowerPC implementations, the 601 and 603 microprocessors.

This example is a short sequence that multiplies two floating-point values together in a loop body. The scheduler interleaves the loads and integer instructions for the 603 because the 603 has a load/store unit that is separate from the integer unit, and it can therefore issue the integer and load instructions together in one clock cycle. Load instructions flow through the integer unit of the 601, and cannot be issued with other integer instructions. For this reason, the scheduler places the loads first in the sequence for the 601 so the results of the load will be available as soon as possible for the multiply instruction.

Memory hierarchy. The complexity of the load/store units, primary and secondary caches, and main memory have strong effects on the performance of any microprocessor. It is important for the compiler to carefully arrange memory operations in order to maximize overall performance. There are several areas in which the compiler can make substantial improvements in the memory performance.

Scientific code is likely to see the biggest effects of cache optimizations by the compiler. The abundance of array references in typical scientific code presents patterns of memory accesses which the compiler can analyze. The compiler can then perform

such optimizations as loop interchanging and cache blocking. These types of transformations can change the patterns of memory accesses to better utilize the data cache. The transformations applied by the compiler to the code depend on the size and organization of the caches, which vary across implementations. Sometimes, just skewing the starting address of large data objects in memory can make a drastic difference in cache behavior and overall performance.

One other method the compiler can use to help improve cache performance is data prefetching [6]. The PowerPC Architecture provides a data cache block touch instruction that will tell the processor to try to load the data at the specified address into the data cache [5]. It will not cause a system error handler to be invoked, and is therefore safe to issue even if that data turns out not to be used. On loop-based code, the touch instruction can be used to eliminate cache miss latencies by overlapping the load time with other computations before the data is actually accessed. A data cache block touch for store instruction is also available which can take exclusive ownership of a block in a multiprocessing environment.

For highly superscalar processors, control flow instructions can quickly become the bottleneck in instruction issue and throughput. There are several areas in which the compiler can help reduce this impact. One area is by performing transformations such as loop unrolling. This reduces the number of branches taken during the execution of a loop, and also makes the code from several iterations of the loop available to the compiler to schedule. This provides opportunities for other types of optimizations that might not exist with just one iteration of the code.

The compiler also provides prediction hints to processors about whether branches are likely to be taken or not taken. The 601 and 603 microprocessors look at the prediction bit of the branch while the instruction is loaded into the instruction queue but before it is executed, in order to load the instructions down the predicted path into the instruction queue. In the event that the

branch is not resolved by the time the branch is executed, the instructions in the predicted path are speculatively issued. If the prediction is wrong, then those speculative instructions are cleared, and the instructions from the other path are loaded and issued. This backup can take several clock cycles, impairing performance. Branches that didn't need to be predicted because they were resolved in time and branches that are correctly predicted are desirable because they don't introduce bubbles or wasted clock cycles in the pipelines of the microprocessors. The compiler tries to schedule branches so the condition on which it depends will be determined before the branch is executed. In those cases where there are not enough instructions to schedule the condition and the branch apart, the compiler uses several branch prediction heuristics to set the prediction bit of the branch.

Another method to lower the impact of branches in compiled code is to eliminate them altogether. The PowerPC Architecture has a powerful set of bit-field rotation and manipulation instructions. These instructions can sometimes be used in sequences of code that would traditionally require a branch instruction. For example, obtaining the result of the comparison of a value with zero is traditionally performed through use of a conditional branch in conjunction with two possible assignments, depending on the result of the comparison. In the PowerPC Architecture, such a value can be obtained without the use of branches as follows:

```
cmpi    crX,rS,0
beq     crX,$+8
addi    rD,r0,1
```

becomes

```
cntlzw  rS,rS
addic   rS,rS,-32
rlwinm  rD,rS,1,31,31
```

Memory space taken up by code and data as well as the run-time memory needed for stacks can become very important issues in some environments. The compiler has several options that optimize for code

and data space. For instance, function prologue and epilog code are responsible for saving and restoring any callee saved registers in order to preserve the original values in the registers. One space-saving option is to use subroutines that save and restore the registers instead of individual instructions in the prologue and epilog. This replaces several instructions with one function call which saves code space. The trade-off is that the functions have overhead associated with them that may cause lower performance. Other optimizations that may help performance can increase space, such as loop unrolling and function inlining. These kinds of optimizations can be turned off, while keeping other optimizations active.

An optimization to save space in the data sections of programs is to sort the data and store the data objects by size. This can eliminate wasted space that might otherwise be needed for padding in order to keep data objects on proper alignment boundaries. The PowerPC Architecture does not require data references to be aligned, but performance may be drastically reduced if unaligned references are made. There is a document under the working name of the PowerPC Embedded Systems Specification. This specification is defining a standard for embedded systems and puts a much higher emphasis on space saving than the PowerOpen ABI. Changes in the areas mentioned, and other changes such as the stack definition to decrease the amount of memory needed are defined for systems in embedded systems areas.

Summary

Motorola has developed a modular approach to generating highly optimizing PowerPC compilers for a broad range of target and host environments. By leveraging a common set of intermediate representations and optimization routines, the compilers can be quickly customized for individual PowerPC implementations, including the 601, 603, 604 and 620 microprocessors, as well as embedded control versions of the architecture currently under design. Ongoing efforts to integrate the compilers with simulators, debuggers

and programming environments will provide additional mechanisms through which the modular design of the compilers can be exploited to reduce time to market for PowerPC systems and application developers in all segments of the computing industry. **C**

References

1. Anderson, W. An overview of Motorola's PowerPC simulator family. *Commun. ACM* (June 1994).
2. Diep, T., Shen, J. and Phillip, M. EXPLORER: A retargetable and visualization-based simulator for superscalar processors. In *Proceedings of the Twenty-Sixth International Symposium on Microarchitecture*, Austin, Tex., Dec. 1993.
3. *PowerOpen Application Binary Interface*. Draft for First ed., IBM, Oct. 1993.
4. *PowerPC User Instruction Set Architecture*. Book I, Version 1.04, May 4, 1993.
5. *PowerPC Virtual Environment Architecture*. Book II, Version 1.04, May 4, 1993.
6. Software prefetching. *SIGPLAN Not.* 26, 4 (Apr. 1991), 40-52.

PowerPC, PowerPC Architecture, PowerPC 601, PowerPC 603, PowerPC 604, PowerPC 620 are registered trademarks of IBM.

About the Authors:

JULIE SHIPNES is currently a senior design engineer in the RISC software group at Motorola in Austin, Texas. Current technical interests include compiler optimization techniques, microprocessor performance analysis and high-performance graphics applications.

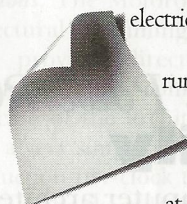
MIKE PHILLIP is currently the manager of the compiler and tools development team in the RISC software group at Motorola in Austin, Texas. Current technical interests include parallelizing compilers, microprocessor performance analysis and microarchitectural simulation. **Authors' Present Address:** Motorola RISC Software Group, Mail Drop OE112 6501 Wm. Cannon Drive West Austin, TX 78735; email: { julie, phillip }@oakhill.sps.mot.com; @oakhill.sps.mot.com

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/94/0600 \$3.50

BE LESS PRODUCTIVE AT THE OFFICE.

The office has always been a place to get ahead. Unfortunately, it's also a place where a lot of natural resources start to fall behind. Take a look around the next time you're at work. See how many lights are left on when people leave. See how much paper is being wasted. How much



electricity is being used to run computers that are left on. Look

Use both sides of the paper when writing a memo.

at how much water is being wasted in the restrooms. And

how much solid waste is being thrown out in the trash cans. We bet it's a lot.

Now, here are some simple ways you can produce less waste at work.

When you're at the copier, only make the copies you need. Use both

sides of the paper when writing a memo. Turn off your light when you

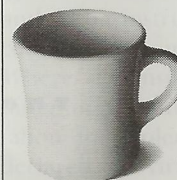
leave. Use a lower watt bulb in your lamps. Drink your coffee or tea out

of mugs instead of throwaway cups.

IT'S A CONNECTED WORLD. DO YOUR SHARE.

Set up a recycling bin for aluminum cans and one for bottles. And when

you're in the bathroom brushing



your teeth or washing your face,

don't let the faucet

Drink out of mugs instead of throwaway cups.

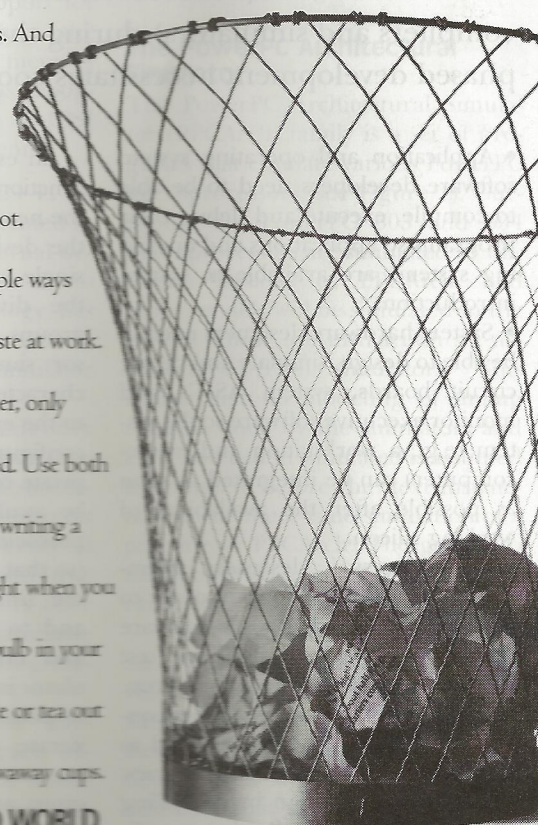
run. Remember, if we use fewer resources

today, we'll save more for tomorrow.

Which would truly be a job well done.

FOR MORE INFORMATION AND TIPS

CALL 1-800-MY-SHARE



A Public Service of This Publication



The Making of the PowerPC



An Overview of Motorola's PowerPC Simulator Family

The successful introduction of a new computer architecture into the marketplace requires that both software and hardware be available simultaneously at the time of system introduction. Moreover, there is substantial need for software tool support (e.g., compilers and simulators) during the design phase of the microprocessor itself. Such phased development necessitates coordination among several groups:

- Application and operating system software developers need to be able to compile, execute and debug their programs so applications and operating systems are available at system introduction.
- System hardware designers need to be able to design, simulate and debug circuit boards, system ASICs and monitor executive software so the system (e.g., a workstation and microcomputer) can be integrated as soon as possible after the availability of working silicon.
- Microprocessor designers themselves need software tools in order to write and debug diagnostic software used to debug chip designs and test the functionality of the final silicon.
- Prospective customers (either systems vendors or end users) need to be able to "kick the tires" of a new microprocessor design by compiling well-known or application-specific benchmark code and running it on a simulator in order to get estimates of system performance.

An essential part of each of these functions is a software simulator of the new microprocessor(s) that solves the designer's needs. Currently, no single simulator program can solve the diverse needs of these user groups. However, any microprocessor simulator should have several characteristics that enhance its utility to the end user, such as being highly configurable (in order that an appropriate configuration of the simulator be available to solve the particular problems of the end user), extensible (so that the end user's effort in using the simulator has maximum effect and to avoid unnecessary redesign and "creeping featurism" by the simulator implementers), and as efficient as possible (to allow the end user to iterate through as many compile-execute-debug cycles as possible in a given amount of time and to minimize the time necessary to run a benchmark).

Motorola and IBM are currently developing the PowerPC family of

RISC microprocessors [7, 8, 9]. To support this effort, Motorola's RISC software team is developing new software tools, including state-of-the-art compilers and simulators. This article describes three members of Motorola's PowerPC simulator family. All of these simulators are based on a common set of object-oriented software technology.

Features of Motorola's PowerPC Simulators

All members of Motorola's PowerPC simulator family share some common features. This common feature set ensures that support code developed on one simulator will run appropriately on another member of the family. It also lessens the burden of learning a new set of commands for a new simulator.

Command line interpreter. All of Motorola's PowerPC simulators use Tcl [6] as a command line interpreter (CLI) language. Tcl (a "tool com-

mand language" which is pronounced "tickle") is an efficiently parsed language created by John Ousterhout of the University of California at Berkeley. Tcl offers excellent facilities for string manipulation and program extension. Many of the system support utilities are implemented in Tcl.

Tcl allows the user to define variables and subroutines, implement flow control and in general adapt the user interface to the particular needs of the user. Tcl allows tedious sequences of simulator commands to be replaced by parameterized command shortcuts. Also, the simulators try to make as few policy decisions as possible. Instead, these policy routines may be implemented as Tcl subroutines. Then, these routines are invoked as a callback routine from the simulator command interpreter. In this way, the end user may implement arbitrarily complex task handler routines for managing simulator events or error conditions.

A distinct advantage to Tcl as a CLI language is the availability of the Tk extension to Tcl. Tk is a widget library that works with the X Window System in order to facilitate rapid development of graphic user interfaces (GUIs). Motorola's PowerPC architectural and timing simulators will be extended with user-extendable GUIs based on the Tk toolkit in future versions.

There is also a very easy-to-use alias facility (also implemented in Tcl) which allows an end user to use one-word abbreviations for longer commands, much like the alias facility availability in some Unix shells.

Operating system emulation mode.

All of Motorola's PowerPC simulators were written to simulate the entire virtual machine presented by the corresponding PowerPC microprocessor. This in particular allows the end user to boot, debug and analyze the performance of an operating system kernel. This capability requires that an appropriate kernel be compiled with the correct device drivers and that corresponding external device simulation modules have been written and dynamically linked into the simulator.

Booting a kernel is an excessive

amount of work if the goal of simulation is to evaluate the performance of user-level programs. Therefore, the Motorola PowerPC simulators have the ability to intercept the system call trap and to vector the simulator code to a predefined set of subroutines which emulate the operating system trap interface. The simulators come with an emulator for AIX (Advanced Interactive eXecutive, IBM's version of Unix). However, this OS emulation module may be replaced by the end user by means of a user-supplied trap table and trap emulation code which is linked at run time by means of a shared object file.

Support for multiprocessing configurations.

The Motorola PowerPC architectural and timing simulator families provide direct support for multiprocessing configurations. In the case of the architectural simulators, a very simple nonpipelined one-instruction-per-clock timing model is used for the shared memory bus. The timing simulators use a clock-phase accurate simulation of the shared memory bus to achieve support for multiprocessing. In both cases, the full semantics of the shared memory system (in particular, the bus snooping required to implement a MESI cache protocol) are implemented.

Object-oriented design in C++. All of the simulators are implemented in C++ [4, 5] and have an internal design which follows the Booch methodology [2]. This methodology has improved the reuse of code among all of the members of the simulator family.

C++ and the Booch methods offer several features that facilitate the implementation of this set of simulators. This is especially important when one considers the diversity of microprocessors modeled and that some of these microprocessors offer primitives which are unavailable using more conventional programming languages.

The PowerPC Architecture supports both signed and unsigned 64-bit integers. However, all of Motorola's PowerPC simulators will be hosted initially on 32-bit architecture. Moreover, conventional languages (e.g., C) do not typically offer a 64-bit integer type. C++ allowed us to de-

fine a 64-bit integer class and to define operators on that class that make using 64-bit integers as simple (from a syntactical point of view) as the usual 32-bit integers.

The ability to define an abstract register class has allowed us to largely ignore the differences between 32-bit and 64-bit instructions in the source code of the instruction handlers. Where the instruction handlers do differ in semantics from implementation to implementation, they are declared as virtual functions. The customization of such functions can then be incrementally added and maintained.

The ability to derive support classes for the internal interfaces used in the timing simulators allowed specialized classes (representing microprocessor functional units) to take advantage of the interfaces (and constructors) of the ancestor classes. This feature of C++ simplified the construction of a simulated microprocessor and avoided many of the programming problems associated with command parsing and distribution.

The PowerPC Architectural Simulator

The PowerPC Architectural Simulator (PPCArch) family is a set of programs that emulate various PowerPC microprocessors (see Figure 1). Currently, the PowerPC 603 and 604 microprocessors are simulated by different versions of PPCArch, and other models will be simulated in the future. These are instruction set architecture (ISA) simulators. This means that no modeling of timing effect is done by the simulators and each instruction simulated is assumed to complete before the next instruction starts: there is no simulation of a pipeline in PPCArch. The goal is to run PowerPC object code as accurately and as quickly as possible.

The PPCArch simulators are based on a type of simulator originally developed for the Motorola M88K RISC microprocessor by Robert Bedichek for Tektronix [1]. Bedichek developed a style of threaded code simulator that used a unique C language and assembly-code macro function to emulate each instruction in the 88100. He was able to decode an instruction once and use the de-

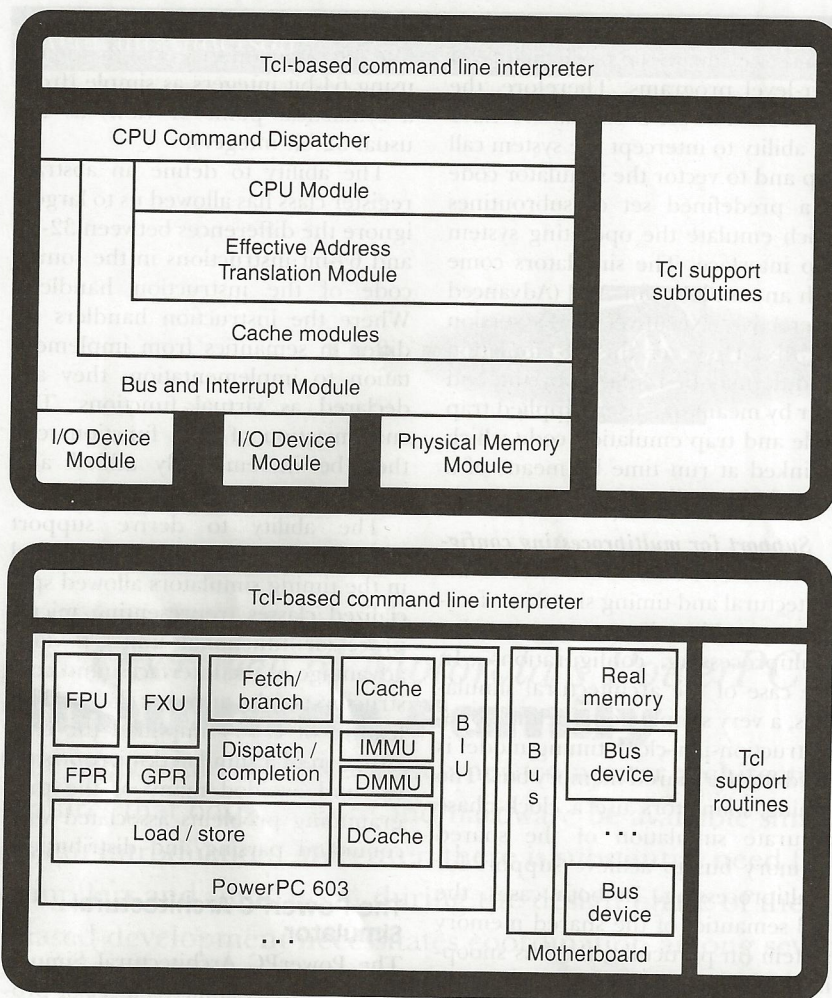


Figure 1. The high-level components of PPCArch

Figure 2. Basic components of PPCSim603

coded form many times, depending on locality of code reference and size of simulated instruction cache. He was also able to simulate the 88K virtual machine sufficiently to boot Unix on the simulator. The performance of this simulator was also very impressive: an average of 130,000 instructions per second when hosted on a 2.5MIPS 68020 Tektronix workstation.

Like Bedichek's simulator, each instruction has its own simulation function (the *instruction handler*) in PPCArch. However, unlike Bedichek we implement our handlers in portable C++ as opposed to assembler macros or nonportable language features. We are willing to trade execution speed for maintainability and

portability as PPCArch will be hosted on multiple architectures. We also cache our decoded instructions, but we use an abstract cache of arbitrary size as opposed to using a model of the actual instruction cache.

The basic design of the physical memory system in PPCArch was also inspired by Bedichek's design. We have a physical memory manager which allocates physical memory pages by an on-demand (i.e., "lazy allocation") scheme. The user is able to allocate a physical memory system of a fixed size without paying the potentially high start-up costs of allocating large chunks of host heap memory. Instead, the physical pages are allocated as they are touched.

An interrupt and bus simulation module provides the interface between the physical memory manager and the caches. The interrupt/bus manager has the ability to attach at run time an arbitrary number of memory-mapped I/O device simula-

tor modules to specified ranges of physical address. Whenever such an address range is read from or written to, the appropriate memory-mapped handler is invoked with the address of the access, the type of access (read, write or execute) and the data (if appropriate) as arguments. By way of a simple deferred interrupt manager and an interface to the real memory system, the memory-mapped handler can simulate the actions of a memory-mapped DMA device without knowing the details of how the simulator works. In this way, device drivers (that part of an OS kernel which directly manages hardware devices) can be tested by writing simple memory-mapped handlers to simulate the associated devices.

The interrupt/bus module also supports symmetric multiprocessing simulation configurations. In this configuration, cpu, effective address translation and cache modules are instantiated as a unit and associated with the interrupt/bus module. The interrupt/bus module provides support for snooping between the caches and physical memory. This configuration will allow OS developers to debug multithreaded operating systems.

The effective address translation module, similar to the physical memory module, has a lazy allocation scheme for its structure maintenance. However, the effective address translation module's primary function is to maintain and to accelerate the effective → virtual → physical translation and to support the semantics of the PowerPC effective memory system. Both memory systems use sparse inverted tree structures to implement the address searches.

Memory-mapped handlers (similar to the instruction handlers and the I/O simulation handlers) are used to implement traps and memory-resident breakpoints. The breakpoint handlers are set using a user-defined code which is returned to the CLI when the user hits the breakpoint. The user is then free to perform whatever manipulations are appropriate to the breakpoint code returned.

The full virtual machine model is primarily implemented with the instruction handlers and with special

The effective address translation module's primary function is to maintain and accelerate the effective $\rightarrow$ virtual $\rightarrow$ physical translation and to support the semantics of the PowerPC effective memory system.

handler functions that are associated with the supervisor visible control registers. This allows us to treat control registers as uniformly as possible while allowing the diversity of function associated with them.

AIX emulation is accomplished by using a monolithic memory model and a special trap handler which is associated with the system call trap vector. When a system call trap is encountered, control is transferred to the system call trap handler. This handler then examines the appropriate general-purpose registers and emulates the system call using the host operating system. The necessary stack space for the user program is mapped separately using CLI routines. Although not all of the AIX system call interface is provided, provision is made for this set of calls to be extended by the user.

The code for PPCArch (which supports both the 603 and 604 PowerPC microprocessor models) comprises approximately 49,000 lines of non-comment, nonblank C++ code.

The PowerPC 603 Timing Simulator

The PowerPC 603 Timing Simulator (PPCSim603) is the first of a series of simulators that accurately model both the instruction set semantics and the detailed pipeline behavior of a PowerPC microprocessor implementation. The basic design emphasis is similar to the design of the PPCArch simulators, although the organization of the simulator internal to the CPU module is significantly different from the architectural simulators. In particular, the organization of PPCSim603 is based on the microarchitectural features of the actual PowerPC 603 microprocessor, as

opposed to the architecture of the PowerPC instruction set as a whole. Therefore, the structure of PPCSim603 is organized around functional units and the synchronized communication among functional units.

Internal to each simulated PowerPC 603 microprocessor in PPCSim603 are functional units which correspond to the actual functional blocks in a physical PowerPC 603 microprocessor. Each of these functional units is implemented as a C++ class and has a specific procedural interface appropriate to its functional type. This procedural interface comprises its public member functions. For example, the bus interface unit (BIU) will have functional interfaces which allow it to communicate directly with the caches and with the external bus. A unit also derives functional interfaces from the classes from which it is derived. This use of C++ inheritance greatly facilitates the implementation of clock and command distribution networks.

Synchronization of the functional units is accomplished by use of a simulated two-phase clock. A functional unit may make no assumptions regarding its order of invocation with respect to any other functional unit within a clock phase. The clock is initially toggled by the user from a command to the bus. The bus distributes the clock signal to all of its bus devices, which include the memory system and any PowerPC 603 microprocessors attached to it. Within the simulated PowerPC 603, the bus clock signal may be multiplied and then redistributed from the BIU to all remaining clocked simulator functional units. The code within a functional unit (for instance, the code

which might simulate a floating-point unit pipeline) is responsible for its own synchronization.

Any functional unit may receive commands directly from the command line. The basic syntax of this command is: `verb unit arg . . .` where `verb` is either `display` or `modify`, `unit` is the pathname to the functional unit of interest (for instance, `motherboard.cpu3.fpu`) and `arg . . .` is an arbitrary list of arguments which is passed into the functional unit for further interpretation. This simple control command syntax greatly simplifies command parsing. The dispatch network that supports the delivery of these commands is managed transparently to the functional unit implementer. To begin to receive commands, a functional unit is simply derived from the `Commandable C++` class, and its virtual functions `display` and `modify` redefined to suit the particular functional unit.

PPCSim603 has an *event* mechanism which allows the end user to instrument a simulator configuration. Events occur when the simulator changes state. An end user can easily write his or her own program to analyze the events that are generated from a simulation run. Events are generated by predefined code internal to the functional unit modules within PPCSim603. Each functional unit can individually enable or disable generation of events. There is also a global event enable switch within the CLI. Examples of event types include cache hits and misses, bus cycles, instruction pipeline state changes and so on.

The event's value is a character string, with colon-separated `name = value` pairs. For instance, a hypothet-

ical instruction event may have the value:

```
evtype
= instr:opcode
= 0 × 48000000:unit
= fxu:addr
= 0xf000:stage = finish:clock
= 10
```

In this example, there are six **name = value** pairs in the event. In order, they indicate that the event type is an instruction event, the hexadecimal opcode is 0×48000000 , the event takes place in the fixed-point unit, the address of the instruction is $0 \times f000$, the stage of the instruction pipeline indicates that the instruction is finished and that the clock count is 10. This form of event is particularly simple to parse and evaluate in Tcl code.

When an enabled event occurs, a global Tcl procedure called an *event handler* is called with the value of the event. Within this routine, a user may filter events, create histograms, gather and process instruction statistics and so forth. The default event handler merely echoes the value of event string to the command line.

Events have proved to be extremely useful during the development of PPCSim603. Events have been used for internal debugging. Instruction trace formatters have been written using the instruction event as a generation mechanism. Finally, events have been used to verify the timing accuracy of PPCSim603. In this last application, a stream of instruction events is parsed and an instruction timing histogram (using Tcl's associative arrays) is constructed during the simulation of a program. This histogram is then compared to a histogram generated from an instrumented version of the PowerPC 603 hardware models, and the variances were analyzed to detect timing inaccuracies. This verification of timing accuracy would be much more difficult to implement without the event mechanism.

Interactions between the PPCSim603 development and test teams made it clear that error handling, similar to event handling, is a policy decision best left to the end user. Although an error could be considered to be just another kind of event, it was

decided that having a separate handler for events made the most sense for a casual simulator user. Therefore, an *error handler* routine (analogous to the event handler) is called when an error condition arises during a simulation run. Error state information is held in two global Tcl variables (**errorCode** and **errorInfo**), which can be interrogated from within the body of the error handler routine. Errors fall into two general categories:

- an error condition may be related to an inappropriate request to the underlying operating system (for example, the user may try to open a nonexistent file), or
- a simulator command (e.g. **display** or **modify**) is in error.

In either case, the error is analyzed and additional information is presented in **errorCode** and **errorInfo** for use by the user's error handler.

The code for PPCSim603 comprises approximately 45,000 lines of noncomment, nonblank C++ code.

The PowerPC 603 Functional Model

Neither PPCArch nor PPCSim603 directly addresses the needs of the system hardware designer. However, PPCSim603 was carefully designed to preserve the interfaces which are present in a real microprocessor system. In particular, the PowerPC 603 CPU module was driven by a clock distributed from the simulated bus, and the memory system was accessible only from that bus. The adherence to these hardware-oriented interfaces by the PPCSim603 implementers made the CPU code from PPCSim603 especially adaptable to other simulation environments. Indeed, one of the requirements of the PPCSim603 design was that the CPU code could be adapted to an industry-standard hardware simulation environment by means of a software "wrapper." The ability to adapt code to run in a different simulation environment also leverages the many engineer-months of testing that go into bringing a simulator to market.

PPCSim603 was adapted to run under the Cadence programming language interface (PLI) for the Verilog simulation environment [3,

10]. This conversion was accomplished by replacing a particular C++ class (the **Pin** class) with a variant class that implemented the PLI interface. The real-memory system was also adapted to run under the PLI interface. Neither the CPU nor the real-memory system can detect whether its environment is standalone (i.e., PPCSim603) or Verilog.

The CLI code from PPCSim 603 was also included in the module which runs under the PLI interface. This is not strictly necessary, since all of the semantics of the processor are accessible from the **Pin** interface. However, it proved especially convenient to do this, since much of the test code for PPCSim 603 was written in Tcl and this code required the CLI in order to run.

This adaptation of the basic PPCSim603 code from standalone simulator to embedded hardware functional model took approximately three engineer months of effort. Much of this effort was spent in learning the Verilog PLI. This PLI has been developed for the C language, but the PPCSim603 CPU module is written entirely in C++. Although there is a mechanism in C++ for dealing with C externals, the reverse situation is not true. In particular, some of the object-oriented semantics associated with C++ classes (the constructors for static objects) caused some initial problems. This problem was solved by having a C++ **main()** routine call back the Verilog PLI **main()** routine (which was renamed **vmain()** and declared an **extern "C"** function). All compilation was done by the native C++ compiler and all linkage done by the native linker. The initial development was done on an IBM RS/6000 workstation running AIX, and used the xlc C++ compiler.

Approximately 900 lines of C wrapper code were necessary to implement the hardware environment interface. In addition, approximately 1,500 lines of (conditionally compiled) C++ code was added to the source code base for PPCSim603. There are currently no plans to turn this PowerPC 603 functional model into a product, since there are other alternatives which satisfy this market segment. The design constraints that

on was accom-
ing a particular
(class) with a vari-
mented the PLI
-memory system
to run under the
er the CPU nor
system can detect
ment is stand-
603) or Verilog.
m PPCSim 603
in the module
the PLI interface.
necessary, since all
the processor are
e Pin interface.
especially conve-
much of the test
03 was written in
quired the CLI in

of the basic
from standalone
edded hardware
ok approximately
months of effort.
was spent in learn-
I. This PLI has
the C language,
CPU module is
C++. Although
ism in C++ for
ternals, the reverse
e. In particular,
oriented semantics
+ classes (the con-
t objects) caused
ms. This problem
g a C++ main()
the Verilog PLI
which was renamed
clared an extern
compilation was
e C++ compiler
one by the native
development was
\$6000 workstation
used the xLC C++
900 lines of C
e necessary to im-
ware environment
on, approximately
tionally com-
was added to the
for PPCSim603.
y no plans to turn
functional model
ce there are other
satisfy this market
gn constraints that

permit a standalone simulator to be adapted to a hardware simulator PLI will prove valuable nonetheless. This constraint will allow Motorola to produce PLI-compliant hardware simulation modules much earlier in the microprocessor design cycle than was previously possible.

Summary

PPCArch and PPCSim603 are the first members of a family of simulators developed by Motorola's RISC software group. They are intended to serve a broad audience of computer system software and hardware engineers.

PPCArch is an architectural simulator that supports both the PowerPC 603 and PowerPC 604 microprocessor models. It is useful to system and application software engineers as a general simulation environment. It has been used to simulate hundreds of application programs for a variety of operating systems, and is being used by OS developers to port OS kernels to the PowerPC Architecture.

PPCSim603 is the first of Motorola's family of cycle-accurate timing simulators for PowerPC microprocessors. It has a number of innovative features, including a user-programmable command line interpreter, a bus interface for user-defined memory-mapped devices, a user-extensible OS emulator and flexible event and error-handling facilities. Using an object-oriented design approach, the CPU core of PPCSim603 was successfully adapted to run in Cadence's Verilog environment with a minimum of additional engineering.

PPCArch and PPCSim603 are the first major projects in M&MTG RISC Software to be implemented using object-oriented design techniques and using the C++ language. The software technology developed in these projects will see much reuse in future Motorola software simulation products. ■

AIX, PowerPC, RS/6000 and xLC are trademarks of IBM.

PPCArch, PPCSim and PPCSim603 are trademarks of Motorola.

Unix is a trademark of AT&T Bell Laboratories.

Verilog is a trademark of Cadence Design Systems, Inc.

References

1. Bedichek, R. Some efficient architecture simulation techniques. In *Proceedings of Winter USENIX*, 1990.
2. Booch, G. *Object Oriented Design with Applications*. Benjamin/Cummings, Calif., 1991.
3. Cadence Design Systems, Inc., *Programming Language Interface Reference Manual for Verilog*. Vol. 1 and 2, 1992.
4. Ellis, M. and Stroustrup, B. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Mass., 1990.
5. Lippman, S. *C++ Primer Second ed.* Addison-Wesley, Reading, Mass., 1991.
6. Ousterhout, J. *Tcl: An embeddable command language*. In *Proceedings of Winter USENIX*, 1990.
7. *PowerPC User Instruction Set Architecture, Book I*, Version 1.04, May 4, 1993.
8. *PowerPC Virtual Environment Architecture, Book II*, Version 1.04, May 4, 1993.
9. *PowerPC Operating Environment Archi-*

ecture, Book III, Version 1.04, May 4, 1993.

10. Voith, R. *The PowerPC 603 C++ Verilog interface model*. In *Proceedings of Spring Compcon '94*, San Francisco, 1994.

About the author:

WILLIAM ANDERSON is presently the manager of the PowerPC microprocessor simulator development team in the RISC software group at Motorola in Austin, Texas. His interests include simulation techniques, computer architecture, computer arithmetic algorithms and VLSI implementation techniques. **Author's present address:** Motorola RISC Software Group, Mail Drop OE112, 6501 William Cannon Drive West, Austin TX 78735; email: wca@bird.sps.mot.com

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/94/0300 \$3.50

BUY RECYCLED.

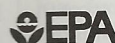


AND SAVE.™

Thanks to you, all sorts of everyday products are being made from recycled materials. But to keep recycling working, you need to buy those products. For a free brochure, call 1-800-CALL-EDF.



A Public Service of
This Publication



ENVIRONMENTAL
DEFENSE
FUND

